University of Crete
Computer Science Department

Subscription Indexes for Web Syndication Systems

Harry Kourdounakis
Master's Thesis

Heraklion, January 2011

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΚΑΙ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΥΡΕΤΉΡΙΑ ΣΥΝΔΡΟΜΏΝ ΓΙΑ ΣΥΣΤΗΜΑΤΑ ΔΗΜΟΣΙΕΥΣΕΩΝ ΠΕΡΙΕΧΟΜΕΝΟΥ ΣΤΟΝ ΙΣΤΟ

Εργασία που υποβλήθηκε από τον
Χαράλαμπος Κουρδουνάκης
ως μερική εκπλήρωση των απαιτήσεων για την απόκτηση
ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΕΙΔΙΚΕΥΣΗΣ

Συγγραφέας:

Χαράλαμπος Κουρδουνάκης, Τμήμα Επιστήμης Υπολογιστών

Εισηγητική Επιτροπή:

Βασίλης Χριστοφίδης, Καθηγητής, Επόπτης

Ιωάννης Τζίτζικας, Επίκ. Καθηγητής

Παναγιώτα Φατούρου, Επίκ. Καθηγητής

Άγγελος Μπίλας, Αναπλ. Καθηγητής
Πρόεδρος Επιτροπής Μεταπτυχιακών Σπουδών

Ηράκλειο, Ιανουάριος 2011

Subscription Indexes for Web Syndication Systems

Harry Kourdounakis
Master's Thesis
Computer Science Department, University of Crete

Abstract

Content syndication has become a popular means for timely delivery of frequently undated information on the Web. It essentially enhances traditional *pull-oriented* searching and browsing of web pages with *push-oriented* protocols in which information publishers deliver brief summaries of the content they publish on the Web, called *news items*, while information consumers subscribe to a number of *feeds* seen as information channels and get informed about the addition of recent items. Today, web syndication technologies such as RSS or Atom are used in a wide variety of applications spreading from large-scale news broadcasting to medium-scale information sharing in scientific and professional communities. However, they exhibit serious limitations for coping with information overload in Web 2.0 since they imply a tight coupling between feed producers with consumers while they do not facilitate users in finding news items with interesting content. In this work, we are proposing an extension of existing web syndication systems with *content-based filtering and tracking facilities*. In this way, users can express their information interests as *keyword-based queries* (rather than subscribing a priori to an entire channel) which will be matched at real time against the streams of news items originating from different feeds.

To efficiently check whether all keywords of a subscription also appear in an incoming news item (i.e., board match semantics) we need to index the subscriptions. Two are the main indexing schemes proposed in the literature, namely, *Count-based* (explicitly counting the number of contained keywords) and *Tree-based* (implicitly counting the number of contained keyword). Unfortunately, the majority of data structures employed in these indexes concern structured subscriptions under the form of sets of attribute-value pairs. In our work, we are interested in indexing unstructured subscriptions formed by sets of keywords according to both Count and Tree-based schemes and study their behavior for critical parameters of realistic web syndication workloads. More precisely, we rely on an ordering of vocabulary terms to build a *Trie* (for tree-based index) that exploits the *covering relationships between subscriptions* compared to the flat search space implied by an *Inverted File* (for count-based index). Then, we experimentally investigate (a) how the morphology of the two indexes is affected by different workload parameters, i.e. the vocabulary distribution, the size of the vocabulary, and the size of the subscriptions and b) how the Trie and Inverted File morphology impacts the scalability and performance of the two indices for realistic characteristics of subscriptions and news items.

The main conclusions drawn from our study are the following. Trie depth is bounded by the size of the indexed subscriptions (in realistic settings is small from 2 up to 6 terms), while its width by the size of the vocabulary of indexed terms (in realistic settings is very large up to 1,500,000 terms). To cope with large vocabulary sizes exhibited in reality we have considered path compression on Trie nodes featuring a unique child. The total number of Trie nodes as well as their morphology is affected by the vocabulary terms' distribution (usually power laws). When the actual terms' frequency in subscriptions follows the initial Trie ordering, subscription factorization will be important resulting to a left deed structured Trie. In the opposite, factorization becomes less important resulting to a right shallow Trie where path compression is now more intense. In particular, when the reverse ordering is followed a great number of subscriptions will be stored at the sub-Tries of the low ranked terms (i.e., the most frequent ones) and thus the pruning of the search space during matching will be significant. Besides Trie

morphology, matching time is also affected by the size of the incoming news items (in realistic settings lies between 5 and 50 terms). In this context, Trie outperforms in matching time from 1 up to 3 orders of magnitude the Inverted File-based index at the expanse of double memory requirements while the Trie throughput rate when indexing with 10,000,000 subscriptions achieves ≈ 500 items/sec (vs ≈ 34 news items/sec for Inverted File).

Supervisor: Vassilis Christophides
Professor

Περίληψη

Η συνεχής αύξηση των διαθέσιμων πληροφοριών στον παγκόσμιο ιστό έχει ως αποτέλεσμα την ολοένα και περισσότερη χρησιμοποίηση τεχνολογιών δημοσιεύσεως περιεχομένου, για την έγκαιρη διάθεση παραγόμενης πληροφορίας. Οι τεχνολογίες δημοσιεύσεως περιεχομένου, στην ουσία ενισχύουν το παραδοσιακό μοντέλο έλξης (pull) που εφαρμόζεται κατά την αναζήτηση και περιήγηση των ιστοσελίδων με πρωτόκολλα του διαδικτύου για δημοσιεύσεις και συνδρομές βασιζόμενες στο μοντέλο ώθησης. Σήμερα, τεχνολογίες για την δημοσίευση περιεχομένου, όπως για παράδειγμα RSS και Atom , χρησιμοποιούνται από μια ευρεία ποικιλία εφαρμογών που κυμαίνονται από μεγάλης κλίμακας μετάδοση ειδησεογραφίας μέχρι μικρής κλίμακας ανταλλαγής πληροφοριών μελών των κοινωνικών δικτύων. Ωστόσο, οι υπάρχοντες τεχνολογίες δημοσίευσης περιεχομένου RSS ή Atom παρουσιάζουν σοβαρούς περιορισμούς για την αντιμετώπιση της υπερφόρτωσης της πληροφορίας στο σημερινό περιβάλλον του Web 2.0, ενώ παράλληλα συνεπάγονται μια δυνατή αλληλεξάρτηση μεταξύ των δυο συμβαλλομένων πλευρών (συνδρομητές και εκδότες). Σε αυτή την δουλειά προτείνουμε μια επέκταση της υπάρχουσας τεχνολογίας συστημάτων δημοσιεύσεως περιεχομένου στον ιστό με υπηρεσίες φιλτραρίσματος βασισμένες στο περιεχόμενο και εντοπισμού. Προς αυτή την κατεύθυνση οι χρήστες θα μπορούν να εκφράσουν ενδιαφέρον σε πληροφορία με επερωτήσεις βασισμένες σε λέξεις κλειδιά (αντί εκ των προτέρων συνδρομών σε ένα ολόκληρο κανάλι) οι οποίες θα αποτιμούνται σε πραγματικό χρόνο πάνω από ροές αντικειμένων ειδησεογραφίας RSS προερχόμενα από διαφορετικά feeds.

Η αποδοτική αποτίμηση ενός εισερχομένου στοιχείου ειδησεογραφίας, που συνιστάτε στον έλεγχο εάν όλες οι λέξεις κλειδιά μιας συνδρομής επίσης εμφανίζονται στο συγκεκριμένο αντικείμενο (λ.χ. broad match semantics), προϋποθέτει την ευρετηρίαση των συνδρομών. Οι βασικές προσεγγίσεις ευρετηρίων που έχουν προταθεί στην βιβλιογραφία είναι οι προσεγγίσεις βασισμένες σε μετρητές (ρητή καταμέτρηση των λέξεων κλειδιών που περιέχονται στο αντικείμενο ειδησεογραφίας) και οι προσεγγίσεις βασισμένες στα δέντρα (έμμεση καταμέτρηση των λέξεων κλειδιών που περιέχονται στο αντικείμενο ειδησεογραφίας). Δυστυχώς η πλειονότητα των δομών που έχουν προταθεί για υπάρχοντα ευρετήρια αφορούν δομημένες συνδρομές αποτελούμενες από σύνολα ζυγαριών πεδίου-τιμής. Στην συγκεκριμένη δουλειά μας ενδιαφέρει η ευρετηρίαση συνδρομών αποτελούμενες από σύνολα λέξεων κλειδιών (αδόμητη πληροφορία) χρησιμοποιώντας και τις δύο προαναφερθείσες προσεγγίσεις και η μελέτη της συμπεριφοράς τους για κρίσιμες παραμέτρους πραγματικών φόρτων εργασίας συστημάτων δημοσιεύσεων περιεχομένου στον ιστό. Πιο συγκεκριμένα, υποθέτουμε μια διάταξη του λεξιλογίου των συνδρόμων για την κατασκευή ενός Trie (προσέγγιση βασισμένη σε δέντρα) που εκμεταλλεύεται τις σχέσεις εγκλεισμού μεταξύ των συνδρομών σε αντιδιαστολή με τον επίπεδο χώρο αναζήτησης που συνεπάγεται από ένα Ανεστραμμένο Ευρετήριο (Inverted File) (προσέγγιση βασισμένη σε μετρητές). Διερευνούμε πειραματικά (α) πως η μορφολογία των δύο ευρετηρίων επηρεάζετε από διαφορετικού παραμέτρους του φόρτου εργασίας λ.χ. την κατανομή του λεξιλογίου, το μέγεθος του λεξιλογίου και το μέγεθος των συνδρομών και (β) πως η παρατηρούμενη μορφολογία του Trie και του Inverted File επηρεάζει την κλιμακωσιμότητα και την επίδοση των δύο ευρετηρίων για πραγματικά χαρακτηριστικά συνδρομών και αντικειμένων ειδησεογραφίας.

Τα βασικά συμπεράσματα που συνεπάγονται της συγκεκριμένης μελέτης ακολουθούν. Το βάθος του Trie οριοθετείται από το μέγεθος των συνδρομών που έχουν ευρετηριαστεί (κάτω από ρεαλιστικές προϋποθέσεις είναι μικρές, από 2 έως 6 όρους), ενώ το πλάτος του οριοθετείται από το μέγεθος του λεξιλογίου των ευρετηριασμένων συνδρομών (κάτω από ρεαλιστικές προϋποθέσεις είναι μεγάλο και φθάνει έως και τους 1,500,000 όρους). Για να αντεπεξέλθει στα μεγάλα μεγέθη λεξιλογίου, όπως παρατηρούνται στην πραγματικότητα, έχουμε θεωρήσει συμπίεση μονοπατιών στους κόμβους του Trie που απασχολούν ένα μοναδικό παιδί. Ο συνολικός αριθμός των κόμβων που απασχολεί το Trie καθώς επίσης και η μορφολογία τους επηρεάζεται από την κατανομή του λεξιλογίου των συνδρομών

(συνήθως ακολουθεί νόμους ισχύος). Πιο συγκεκριμένα, όταν η πραγματική διάταξη των όρων βάση της συχνότητας εμφάνισης τους ακολουθεί την διάταξη που θεωρεί το Trie για κατασκευή, τότε η παραγοντοποίηση των συνδρομών (subscription factorization) στην βάση των κοινών προθεμάτων θα είναι μεγάλη ενώ θα καταλήξουμε να έχουμε ένα αριστερά βαθύ Trie. Στην αντίθετη περίπτωση θα έχουμε λιγότερη παραγοντοποίηση και θα καταλήξουμε σε ένα αριστερά ρηχό Trie, στο οποίο η συμπίεση μονοπατιών θα είναι όμως πιο έντονη. Πέραν της μορφολογίας, ο χρόνος αποτίμησης επηρεάζεται κατά κύριο λόγο και από το μέγεθος του εισερχομένου προς αποτίμηση αντικειμένου ειδησεογραφίας (σε πραγματικό περιβάλλον το μέγεθος τους κυμαίνεται μεταξύ 5 και 50 όρων). Το Trie υπερτερεί του Inverted File από μία έως τρείς τάξεις μεγέθους, χρησιμοποιώντας όμως διπλάσια μνήμη. Για αυτό τον λόγο η ρυθμαπόδοση που επιτυγχάνει το Trie έχοντας ευρετηριάσει 10,000,000 συνδρομές είναι $\approx$ 500 αντικείμενα ειδησεογραφίας το λεπτό (το Inverted File επιτυγχάνει $\approx$ 34 αντικείμενα ειδησεογραφίας το λεπτό).

Επόπτης Καθηγητής: Βασίλης Χριστοφίδης
Καθηγητής

Ευχαριστίες

Στο σημείο αυτό θα ήθελα να ευχαριστήσω θερμά τον επόπτη καθηγητή μου Κ. Βασίλη Χριστοφίδη για την άψογη συνεργασία, το χρόνο που μου αφιέρωσε, και την πολύτιμη καθοδήγησή του. Κατά την διάρκεια των ακαδημαϊκών μου χρόνων κατάφερα να τον γνωρίσω όχι μόνο σαν καθηγητή αλλά και σαν άνθρωπο και τον ευχαριστώ θερμά για τα εφόδια που αποκόμισα κοντά του, που θα αποτελέσουν τις βάσεις για την μετέπειτα καριέρα μου.

Δεν θα μπορούσα να μην αναφέρω σε αυτό το σημείο το Ινστιτούτο Πληροφορικής του Ιδρύματος Τεχνολογίας και Έρευνας τόσο για την υλική όσο και την οικονομική στήριξη που μου πρόσφερε. Επίσης θα ήθελα να ευχαριστήσω θερμά όλα τα μέλη της ομάδας των Πληροφοριακών Συστημάτων του Ινστιτούτου Πληροφορικής - ΙΤΕ για την ευχάριστη συνεργασία και για το γεγονός ότι υπήρξαν όχι μόνο συνεργάτες αλλά και φίλοι. Ειδικότερα, δεν θα μπορούσα να μην ευχαριστήσω προσωπικά την Νέλλη Βουζουκίδου, για το κομμάτι της γεννήτριας συνθετικών δεδομένων που μου έδωσε και τις πολύ εύστοχες συζητήσεις και παρατηρήσεις τις γενικότερα πανω στην δουλειά. Επιπρόσθετα, θα ήθελα να ευχαριστήσω τους Zeinab Hmedeh και Cedric du Mouza για την βοήθεια πάνω στο αναλυτικό μοντέλο του Trie.

Επίσης θα ήθελα σε αυτό το σημείο να ευχαριστήσω προσωπικά τους πολύ καλούς μου φίλους Γιώργο Τσάμη, Μιχάλη Σκάλτσα και Δανιήλ Γαλουζή, για την υποστήριξη και τις καλές στιγμές, που μου πρόσφεραν κατά την διάρκεια των χρόνων αυτών (και όχι μόνο). Επιπρόσθετα θα ήθελα να ευχαριστήσω την Κατερίνα για την αμέριστη συμπαράσταση εμπιστοσύνη, και υπομονή που μου έδειξε τόσο στις εύκολες όσο και στις πιο δύσκολες στιγμές κατά την διάρκεια εκπόνησης της συγκεκριμένης εργασίας.

Τέλος δεν θα μπορούσα να παραλείψω την οικογένεια μου, Τσαμπικό, Άννα και Μαίρη για την θερμή υποστήριξη και εμπιστοσύνη που μου έδειξαν, χωρίς την οποία δεν θα ήταν δυνατή η εκπόνηση της συγκεκριμένης εργασίας.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Web Syndication Systems

With the continuous growth of online information, content syndication has become a popular means for timely information delivery on the Web. It essentially enhances traditional *pull-oriented* searching and browsing of web pages with *push-oriented* publishing formats and subscription protocols of web content. Web syndication initially was aiming to exchange on a day-to-day basis concise summaries of the information published on the Web such as news headlines, search results, 'What's New', job vacancies, and so forth. Today, web syndication technologies such as RSS[1] or Atom[2] are widely used in a variety of applications ranging from large-scale news or social media (blog, wiki, etc.) broadcasting to medium-scale information sharing in scientific[3] and professional[4] communities. Note also that web syndication technologies have recently attracted interest in the context of the mobile web as a streaming protocol for mobile clients.

In the context of RSS/Atom, information publishers deliver brief summaries of the content they publish on the Web, called *news items*, while information consumers using adequate RSS/Atom software subscribe to a number of feeds seen as *information channels* and get informed about the addition of recent items. Each item consists of a *title*, a *description*, and a *link* pointing back to the actual source of information, such as a news or a blog site. Additional metadata regarding the published information also exist such as the *author's name*, the *publishing date*, the *written language*, etc. In essence, an RSS feed is an URL that returns an XML document in an accepted RSS format, possibly with informal additions. Atom is an effort of IETF to come up with a well-documented, standard syndication format. Although RSS preceded ATOM, the former still achieves wide acceptance, despite the fact that the latter is the only one of the two that has an active working group supporting it and has been adopted by major companies involved in the area of web data management such as Google (GData)[5]. Since both standards provide the same basic functionally, the term 'RSS' is commonly used to also refer to Atom syndication.

Every time new information content becomes published, a new item is appended to the specific XML file, encoding an RSS feed. An example of an RSS feed document is illustrated in Figure 1.1.

Once a feed is available, specific software can regularly fetch the RSS/Atom file to get the

[1] web.resource.org/rss/1.0/

[2] tools.ietf.org/html/rfc5023

[3] For instance, biologynews.net/rss.php or ubio.org/index.php?pagename=ubioRSS

[4] For instance,rss-specifications.com/finance-rss-feeds.htm

[5] code.google.com/apis/gdata

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0">
  <channel>
  <title>BBC News | Latest Published Stories | UK Edition</title>
  <link>http://news.bbc.co.uk/</link>
  <description>Updated every minute of every day</description>
  <item>
    <title>Hearing first-hand about the 2014 Handover Ceremony</title>
    <link>http://news.bbc.co.uk/go/rss/-/1/hi/school_report/student_reports/9190806.stm</link>
    <description>
      School Reporters from Shawlands Academy in Glasgow found out all about the 2014
      Commonwealth Games Handover Ceremony from one of the students at their school
      who was there.
    </description>
  </item>
  </channel>
</rss>
```

Figure 1.1: Example of RSS version 2.0 XML document

most recent items on the list and presented them in a readable manner to the user. This is the role of dedicated desktop (or web) applications known as *RSS readers* or *aggregators*. The user specifies the feeds to which he/she wishes to subscribe by specifying their corresponding URLs. Once subscribed to a feed, the RSS reader is able to check for new items at user-determined intervals and retrieve the update. Examples of RSS readers are amongst others FeedReader[6] and RSSOwl[7]. An example of a web based RSS reader is that of GoogleReader[8].

More and more official news sites rely on web syndication technologies to inform potentially interested users in a timely manner. Usually, news sites define a broad set of information categories (e.g. economy, jobs, sports, etc.) and establish an RSS channel for each one of them. Whenever a news item becomes available, the corresponding channel is updated with the RSS item referring to the specific news article. News consumers are asynchronously informed about news items, as soon as they are published by their producing source, or broadcasted by news infomediaries.

In conjunction with their original usage for delivering news over the web originating from authority sources (e.g., news agencies, newspapers, etc.), web syndication popularity has been recently increased also due to the high volumes of user generated content, generated in various social media (blogs, wiki, etc.). As opposed to the traditional news setting, where the roles of information producers and consumers is clearly separated, these roles are actually blurred in the setting of citizen journalism in social media. For example, with the use of web blogs anyone can become a publisher and post content on the web. Another example is social media such as Facebook[9] and Twitter[10]. In such web applications, users regularly read from and post to their community of friends small messages whose content spans from everyday activities a person might be involved in, to subjective opinions on social matters and politics. Various forms of social media is actually syndicated and delivered outside the social media applications via RSS/Atom standards.

In this context, users may be overwhelmed by an important amount of information actually

[6] www.feedreader.com
[7] www.rssowl.org
[8] www.google.com/reader
[9] facebook.com
[10] twitter.com

delivered by web syndication systems. This can be a result either of users' subscription to a large amount of different channels simultaneously, or to frequently updated channels or even to both. For instance, to get informed about todays economic crisis a user should subscribe to economy, politics and business information channels around the world. In order for the user to cope with the large amount of delivered news items, many RSS readers provide searching functionality that filter the available news items according to particular user interests. FeedReader[11] for example allows users to assign a set of keywords (terms) to a specific RSS feed. Whenever a new item is fetched for that specific feed, the application tests if all of the terms specified are also present in the item, and notifies the user accordingly.

Recently, several mediator services have emerged providing filtering and management functionality over RSS feeds. xFruites[12] for example enables users to aggregate and transform RSS feeds into other content types such as PDF. With the use of FeedRinse[13] the user can automatically filter out syndicated content that is not of interest to him/her. Filtering is performed at a keyword basis on the set of feeds the user specifies. For the more experienced users, Yahoo Pipes[14] offers a graphical mashup development environment that serves as a composition tool for aggregating, manipulate, and filtering syndicated content.

However, with the usage of RSS readers and management services, a user gets notified only about items belonging to the channels she/he has already subscribed to. The existence of any particular RSS channel must therefore be known beforehand. This tight coupling interaction model gives of course to the end user a full control over what content finally gets delivered to him/her, at the expanse of a priori knowledge of the information channels that might potentially be of interest to her/him. Clearly, this is feasible only when the number of channels is limited to few authority sites of news agencies and newspapers, and not to a myriad channels bounded to personal blogs and social applications. In the news domain, Google News and Yahoo! News are currently tracking 4500 and 5000 sources respectively [29]. For blogs, an event detection system would have to deal with millions of blogs as sources. In reality, however, most users are only interested in a small subset of news articles and blog posts. For this reason, existing web syndication technology cannot effectively address the needs for accurate and timely delivery of information published in social media.

1.1.1 Web News Search Engines

Nowadays few web applications have emerged providing search services for news articles published mainly by authority news sites. Two typical examples are Google News Search[15] and Yahoo News Search[16]. These applications periodically crawl news agencies and newspapers' sites for new articles and append them to the corpus of news items they maintain locally. Users in turn issue *short-living* keyword based queries, which get evaluated over the set of stored items. It should be stressed, that a similar approach also emerged for searching news articles from blogs and other citizen journalism sites such as Google Blog Search[17].

Web news search engines play essentially the role of infomediaries between producers and consumers of news items, but unfortunately rely on a traditional pull model for searching and browsing web content. However, rather than a coarse search on a news source/feed level as

[11] www.google.com/reader
[12] www.xfruits.com
[13] www.feedrinse.com
[14] pipes.yahoo.com
[15] news.google.com
[16] news.search.yahoo.com
[17] blogsearch.google.com

Figure 1.2: Overview of Publish/Subscribe

supported by RSS aggregators and RSS management services, web news search engines operate upon individual news items. In addition, since they are able to crawl not previously known news sources, web news search engines provide a transparent access to news items regardless of whether the users are aware or not of their origin. The major shortcoming of web search engines is related to intellectual property rights that have to be respected in face of different information producers in order to host locally the news items. Furthermore, users have to periodically re-issue their interests under the form of short- living queries every time they want to be informed about recent news items. This may also incur delays in information dissemination when news items are published in a bursty rate (e.g. during environmental or social crises).

A more attractive solution is proposed by publish subscribe systems handling *long-lasting* user subscriptions against which the incoming news items are matched in a streaming mode.

1.1.2 The Publish/Subscribe Interaction Paradigm

A system employing the Publish/Subscribe paradigm consists of publishers, subscribers and a notification service. Publishers generate content in the form of events (e.g. an incoming news item). Subscribers specify their interest in specific events under the form of subscriptions. The notification service serves as an intermediate and is responsible for delivering published events to the appropriate subscribers whose subscriptions are satisfied. Subscribers typically receive only a subset of the total events published. The process of selecting the correct subset of subscriptions each event satisfies, is referred to as *event matching*. The component of the notification service responsible for performing the matching of events is called *event processing engine*. At the 'heart' of this engine there exists one or more *index structures* responsible for storing subscriptions against which the incoming events will be matched.

As we can see in Figure 1.2 one of the main benefits of the Publish/Subscribe messaging paradigm is that it provides *decoupling* between information providers (publishers) and information consumers (subscribers). Information generation and information consumption are performed independent to each other. This decoupling holds in both *space* and *time*. Subscribers are not aware of the existence of publishers and vice versa. In addition, both parties do not have to be active at the same time for event delivery to occur. Publishers can generate content when subscribers are disconnected. In a similar manner, subscribers can be notified about events satisfying their interests when publishers are disconnected.

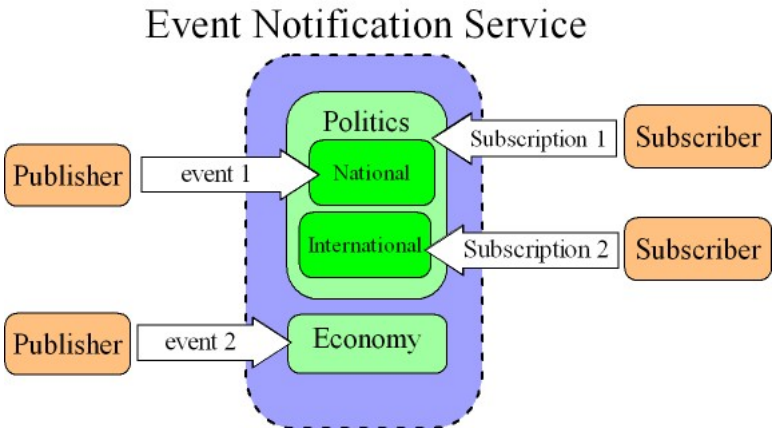


Figure 1.3: Topic-based Publish/Subscribe

The matching semantics adopted by the event processing engine conforms to the subscription scheme a particular Publish/Subscribe system adopts and may differ in the level of expressiveness it supports. Depending on the way events are matched a subscription scheme can be either topic or content based.

Topic-Based Publish Subscribe

Topic based subscription schemes where the first to be proposed in the literature for Publish/-Subscribe systems. In such a scheme, every event is assigned to a particular topic (e.g., politics, economy, business). Topics are pre-defined, and users express their preferences by selecting a subset of the topics that are of interest to them. Every time a particular event is published to the system, subscribers whose subscriptions contain the topic assigned to the published event will be notified.

Topics can be organized in a hierarchy of sub-topics. As we can see in Figure 1.3 users subscribing to a topic will be notified about events matching to the specific topic but also any of its sub topics in the hierarchy. The system defines two topics namely, 'Politics' and 'Economy'. The former is further organized into sub topics 'National' and 'International'. The subscriber interested in 'Politics' will get notified about $event1$ since 'National' falls under 'Politics'. Subscriber 2 on the other hand will not be notified. Typical examples of Publish/Subscribe systems following a topic based subscription scheme amongst others are Corona [36] and Scribe [38].

In a topic based subscription scheme, event matching is rather straightforward. The event processing engine should maintain a mapping between topics and subscriptions: for every defined topic, the list of corresponding subscriptions would be accessed. Every time a new event is published to the system, the event processing engine would have to retrieve the subset of subscriptions corresponding to the topic the particular event is assigned to, and notify each corresponding subscriber about the event. Existing RSS/Atom technologies are essentially a simplified version of the topic based subscription scheme where each channel information provides information about a topic or a set of topics.

Content-Based Publish Subscribe

This type of Publish/Subscribe systems use a subscription scheme based on the content of the published events. Depending on the underlying data model used to represent the content

(structured or unstructured) of events, such Publish/Subscribe can be further specified as either *attribute* or *term* based.

In the attribute based scheme all events conform to a global event schema (like a universal relation) which specifies the number and type of allowed attributes. Each subscription is expressed as a boolean query over the global event schema. More specifically, subscriptions consists of one or more atomic predicates of the form $Attr\ \theta\ Val$, where θ denotes the operator (i.e. $=, <, >$), upon which comparisons are performed. An event is said to match a subscription if and only if for every attribute binding of the subscription the corresponding predicate is satisfied [4].

To better demonstrate how the attribute based subscription scheme applies consider the example of Figure 1.4 with a Publish/Subscribe system for disseminating stock quotes. Whenever a stock quote gets updated an event is published to the system consisting of the following two attributes: the *name* of the quote and its current *value*. Let $S_1 = \{name =' NWK'\ AND\ value > 55\}$ and $S_2 = \{name =' RBS'\ AND\ value > 20\}$ be two subscriptions submitted to the system. Further suppose that a stock update initiated the following event $e = \{name =' RBS', value = 60\}$. Matching event e against the set of subscription S_1, and S_2 would result to S_2 being notified since event e satisfies both predicates of S_2 (but not subscription S_1).

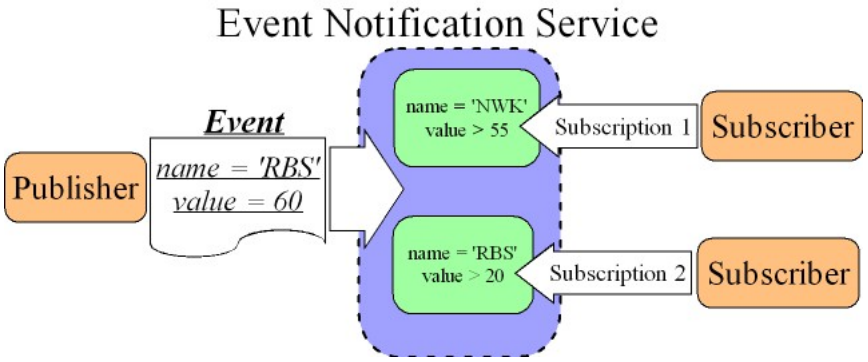


Figure 1.4: Content-based Publish/Subscribe

The term based subscription scheme in turn is more appropriate when the content of published events is unstructured usually taking the form of textual documents. Subscribers in this schema express their interest to particular events via a keyword (term) based subscription language. More than one terms can be logically combined to form boolean subscriptions. Matching semantics depend on the logical operators the subscription language supports (i.e. conjunction, negation, disjunction, etc). For example, under conjunctive semantics, a subscription will match an event if and only if *all* of its terms also appear in a specific event. Consider for instance, a subscription containing the terms 'greece' and 'crisis'. Then, only events whose content contains *both* of the terms 'greece' and 'crisis' will satisfy the subscription and will be finally distributed to the respective subscriber.

It is obvious that content-based is more expressive than topic-based Publish/Subscribe systems as subscriptions can be considered as complex topics specified on demand: each different combination of predicates can be seen as single dynamic topic. This enhanced expressiveness however, introduces a greater complexity with regard to event processing. From an algorithmic viewpoint, the problem of matching is as hard as the partial match problem [24]. Hence, efficient and scalable index structures and algorithms that exploit specific target application domain characteristics should be devised to address the matching problem. Depending on how matching is applied two major approaches exist, namely the *count* and *tree* approach.

A count based approach maintains a counter for each subscription. The purpose of a counter assigned to a subscription S_i is to facilitate matching by recording the number of predicates satisfied by a current event. Given an event, the algorithm iterates over each of the attributes of the event. For each such attribute, it finds the set of subscriptions whose corresponding elementary predicate is satisfied by the attribute value of the event and increments the counter for these subscriptions by one. Having tested all of the attributes of event e, the algorithm returns all the set of subscriptions whose corresponding counter is equal to the number of predicates they contain.

In a tree based approach subscriptions are organized into a rooted search tree. In this matching tree, each node corresponds to a test on some of the attributes and edges are results of such tests. Each node found within a lower level is a refinement of the tests performed in higher levels. Subscriptions are stored at leaves. When evaluating against an event, a top down traversal of the matching tree built over the predicates of the subscriptions is performed. At each node, the test prescribed by the node is executed and the edges consistent to the result are followed (more than one edges might be followed). This is repeated until the leaves of the tree are reached. The set of subscriptions that correspond to the leaves upon which the algorithm has concluded are reported as matched [4].

The advantages of the content based subscription scheme has lead to the realization of a vast number of Publish/Subscribe systems within the scientific and industrial community. Examples of such systems that employ an attribute based model for events are amongst others SIENA [12], Gryphon [4] and Le Subscribe [33]. An example of a term based Publish/Subscribe system used for the selective dissemination of text documents is that of SIFT [46]. Another example is that of COBRA, a system that crawls, filters, and aggregates vast numbers of RSS feeds, delivering to each user a personalized feed based on their interests expressed as term based subscriptions [37]. Finally, YFilter [16] considers the massivelly adopted semi-structured model of XML for events, and aims in providing on-the-fly matching of XML encoded content to larger numbers of query specifications.

1.1.3 Quantifying Performance Targets of Web Syndication Systems

According to a recent experimental evaluation of the characteristics of 8,135 news feeds over an 8 month period [21], RSS/Atom items combine information from both authoritative (e.g. news agencies and journals) and social sources (e.g., blogs or micro blogs), provide only a short overview of the available web content (smaller than advertisement bids and social posts or comments), they are by no mean interlinked (e.g., as Web page hyperlinks or post replies) while the globally observed feeds activity is significantly inferior to the activity observed inside social media applications involving a much large number of users. More precisely, the main conclusions drawn from this experimental evaluation are:

1. 3.6% (which accounts for 292 news feeds of the total number of harvested RSS/Atom feeds had an hourly update (>1 item per hour published) while the corresponding feeds account for the 73.07% of the items published during the period of study. In addition, feeds with a low publication rate exhibit more frequently bursts with an important strength. No major variation in the behaviour of the feeds activity has been observed during the 8 month period of the study. Furthermore, authors haven't observed any strong correlation between feeds subject and their activity.

2. RSS/Atom items are intrinsically semi-structured. The most popular tags are the textual *title* (with an average length of 6.81 terms) and *description* (with an average length of 45.56

terms). Thus the average length of items is 52.37 terms clearly greater than web queries (2-3 terms [39, 47]), advertisement bids (4-5 terms [26]) or micro-blogs ($\approx$ 10 terms), but certainly smaller than blog posts (200-250 terms) or Web pages (450-500 terms excluding tags [28]).

3. The size of the vocabulary employed in RSS items is extremely big. Authors in [21] report a total number of 1,537,730 terms employed by the English feeds of their testbed. For the 5000 most popular terms, 91% correspond to English words (i.e. terms belonging to WordNet[18] dictionary). This ratio decreases linearly to reach less than 50% after rank 20000. However, the weekly vocabulary does not exceed 176,600 terms (of which 85,600 are English terms) with 81,600 (of which 31,400 are English terms) of them being already used in the previous day. Finally, named entities and typos are much numerous than common English terms: a sample of terms with a unique frequency gave 60% of typos among the terms non appearing in WordNet.

However, to the best of our knowledge, no research has been conducted on the characterization of the statistical properties of term-based subscriptions. This is related to the fact that term-based web syndication is still in its infancy and not many commercial systems supporting such a subscription scheme exist. One such example, is Google Alerts[19], however information regarding user subscriptions is considered proprietary and it is not available for statistical analysis. In the rest of our work we will project to term-based subscriptions the experimental finding concerning related web technologies such as Web pages and queries [14, 39, 47, 7, 43, 28], textual advertisements and sponsored search [23, 2, 18, 26]. With regard to the anticipated subscription size, we expect that on average the number of terms a subscription would contain will fall between the range of web queries (2-3 terms [39, 47]) and advertisement bids (4-5 terms [26]). Given the nature of news/blogs, many of the keywords employed by users in queries are event-related [30, 40] .Hence, we expect that the vocabulary size of subscriptions to be comparable to the vocabulary of news items and follow the same term frequency distributions. In addition, we anticipate, that term burstyness in subscriptions follow the burstyness of terms in items (users are interested in news items concerning major events as long as they are published).

1.2 Problem Statement & Contributions

As we have seen, RSS or Atom syndication technologies exhibit serious limitations for coping with information overload in the context of Web 2.0 while they imply a tight coupling between feed producers with consumers. In this thesis we are interested in supporting the functionality of news search engines according to a push model. A naive solution to this end is to handle user queries as long lasting keyword based profiles and evaluate them every time the corpus of indexed news items was updated. It becomes evident, that the naive approach is neither efficient nor scalable given the high publishing rate of news items and the large number of profiles that need to be maintained for real scale web applications. For this reason we advocate a Publish/Subscribe messaging paradigm which implies to efficiently handle *long-lasting* user subscriptions against which the incoming news items are matched in a streaming mode. In particular, we will like to enable users to express their interests as keyword-based queries (rather than subscribing to an entire channel) which will be matched at real time against incoming news items (i.e., events in our setting) from different web feeds. News items are in turn represented as a set of terms

[18] wordnet.princeton.edu
[19] www.google.com/alerts

(i.e. keywords) and ignore the semi-structured tags of the involved RSS/Atom XML files (our choice will be justified latter on in this section). In other terms, we are focusing on *term-based* Publish/Subscribe systems for news items published on the web according to the RSS/Atom standards.

Regardless of the employed subscription scheme, our primary concern is to build an event processing engine that is both *efficient* time/space wise and *scalable* in terms of supported subscriptions and news items in a centralized setting (distribution aspects are outside the scope of our thesis). Efficiency refers to the time the notification service needs to match an incoming news item against the set of stored subscriptions. Ideally, a *term-based web syndication system* should be able to handle millions of users' subscriptions, be able to manage high rates of published news items and afford a small response time in order to achieve an *on the fly event processing* which obviates the need for storing locally the news items.

It is obvious that, in order to ensure good performances of a Publish/Subscribe system independently of the setting employed (distributed/central) to implement the notification service, it is required to efficiently solve the problem of event matching. Depending on how matching is decided, for content based Publish/Subscribe systems, two main approaches been proposed in the literature, namely count-based and tree-based. Towards that end, in this work, we are interested in indexing unstructured subscriptions formed by sets of keywords according to both Count and Tree-based schemes and *study their behavior* for *critical parameters* of realistic web syndication workloads.

Contributions

The main contributions of this work are:

- We studied the memory and matching time requirements of both tree and count-based subscription indices. More precisely, we relied on an ordering of the vocabulary terms to build a Trie-based index that exploits the covering relationships between subscriptions to address the scalability concerns when indexing and searching sets of keywords. As opposed to the flat search space implied by the Inverted File structure of the Count index, the hierarchical search space of the Trie index addresses the following two problems exhibited by the Count-based alternative: (a) for frequent terms the search space of subscriptions is extremely large and (b) in order to support broad match semantics (the universal quantifier on the terms of subscriptions) an additional index structure, namely the counter, is required. In addition we tried to bound the matching and memory requirements of the Trie index based on the statistical properties of the vocabulary of terms.

- We implemented the Trie and Count based indices and devised sace-wide optimizations for both. For the Count index the optimization accounts for: (a) the nodes of the postings sets (compact representation) and (b) count structure (2-level indexing). As an optimization for the Trie index we considered: (a) five different node types (depending on the number of subscriptions and children a particular node contains) and (b) path compression (via compact Trie nodes).

- We conducted a thorough experimental evaluation on both indices, considering all optimizations, with synthetic data generated from statistical properties observed in real application settings. More specifically, we were interested in investigating (a) how the morphology of the two indexes is affected by different workload parameters, i.e. the vocabulary distribution, the size of the vocabulary, and the size of the subscriptions and

(b) how Trie and Count morphology impacts the scalability and performance of the two indices for realistic characteristics of subscriptions and news items.

To the best of our knowledge, no previous work that performs a comparative study on the behaviour of Count and Tree based indexing schemes for critical parameters of realistic web syndication workloads exists.

1.3 Thesis Organization

The structure of this thesis is as follows: In Chapter 2 we formally define the matching problem and present the index structures and algorithms used for matching. We instantiate the discussion with Section 2.1 where we present a naive approach to matching. We argue the need for indexing and next, in Section 2.2 present the Count index for which we provide a distribution based analytical model. We discuss it's properties and expose it's basic limitations. In Section 2.3 we propose a Trie based index for storing subscriptions, and provide an upper bound on the memory and matching requirements based on the Pascal's Triangle construct.

Chapter 3 presents our prototype Publish/Subscribe system used for evaluating the index structures. We provide implementation specific details, in a high enough level, to justify several, essential to performance, design decisions for both the Count and Trie indices. The experimental findings of the Trie index when compared to the Count index are presented in Chapter 4. Firstly, in Section 4.1 we investigate the impact of different term distribution cases on the morphology of both indices, next we show the results from the evaluation of the impact of several related workload parameters. In Section 4.5 we measure the scalability characteristics of the Trie index in comparison to the Count index.

Chapter 5 gives the related work in the more general area of event notification systems. Finally, in Chapter 6 we present our main conclusions and some ideas for future work.

Chapter 2

Subscription Indexes

Nowadays a great amount of news content ranging from journal papers and articles to personal blogs until microblogs in social media is being syndicated on the web. Hence, there exists a demand for services that provide search capabilities for news information. Two typical examples are Google News Search [1] and Yahoo News Search[2]. These services periodically crawl for news and enrich their corpus with published content. These search services are essentially *retrospective* in nature as users issue *short living* queries which get evaluated only once over already existing content[22]. As web search engines, news queries are keyword based and adopt conjunctive semantics. Assume a query q_i and let $\mathcal{I}$ denote the news corpus indexed by such a retrospective search system. Evaluating q_i over $\mathcal{I}$ results to $\mathcal{I}_{\mathcal{MATCHED}} = \{I_k : \forall t_j \in q_i \rightarrow t_j \in I_k\}$. That is to say that a particular news item I_k is considered as an answer when evaluating q_i if and only if every term (keyword) of q_i is also contained within I_k. Consider for example the queries $q1 =' greece' \wedge' crisis'$ and $q2 =' greece' \wedge' crisis' \wedge' IMF'$ and a news item corpus $\mathcal{I}$ of only one news item $I_1 =' greece' \wedge' crisis' \wedge' deficit'$. Evaluating q_i over $\mathcal{I}$ would return news item I_1 because both of the terms 'greece' and 'crisis' of q_1 are also present in I_1. However, news item I_1 would not be in the answer set of q_2 since the term 'IMF' does not belong to I_1.

In the publish/subscribe paradigm for web syndication, users submit *long lasting* subscriptions instead of queries. Whenever a news item is published, it gets evaluated against the set of indexed subscriptions and for every matching the corresponding subscriber is notified. Hence, they are *prospective* in nature since a user submitted subscription will be matched to news items encountered in the future. We assume a keyword base subscription scheme with similar conjunctive semantics to web news search engines. A match occurs if and only if all of the terms (keywords) of a particular subscription S_i are also be present in a news item I_k. For example, assume the subscriptions $S_1 =' greece' \wedge' crisis'$ and $S_2 =' greece' \wedge' crisis' \wedge' IMF'$ and suppose we want to match the news item $I_k =' greece' \wedge' crisis' \wedge' deficit'$ against both S_1 and S_2. Since all of the terms of S_1 are also contained within I_k, S_1 will be reported as matched. Subscription S_2 however, will not be contained in the answer set of I_k, since the term 'IMF' is present within S_2 but not in I_k.

Within the context of web news search engines, it is the user queries that get evaluated over a maintained news corpus. When the publish/subscribe paradigm is applied however, then it is the arriving news items that are evaluated against the stored subscriptions. Clearly, the containment relations with regard to what is being evaluated in each case have been inversed. As we can see in Figure 2.1, within the context of publish/subscribe the roles of queries (subscriptions) and

[1] news.google.com
[2] news.search.yahoo.com

documents(news items) has been inversed.

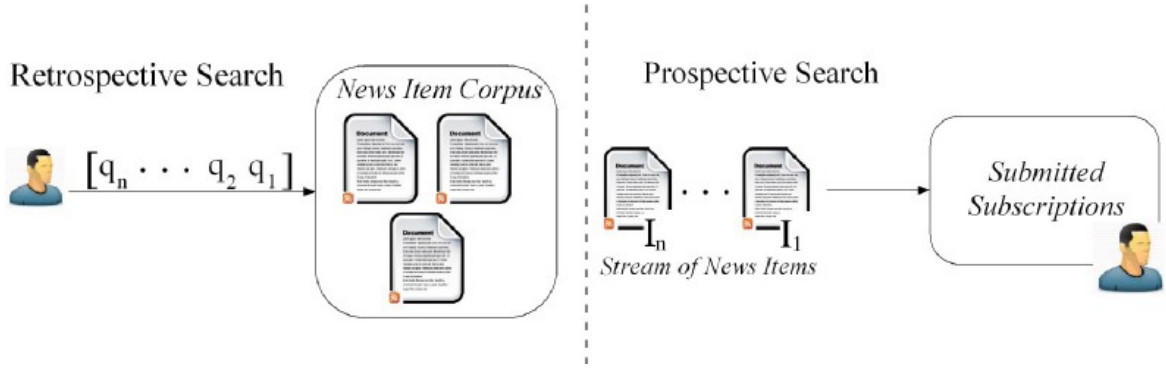


Figure 2.1: Prospective versus Retrospective search

Notation

Consider a vocabulary of terms, $\mathcal{V}_\mathcal{S} = \{t_1, t_2, ..., t_n\}$. In the following we will denote the set of subscriptions submitted to the system as $\mathcal{S} = \{S_1, S_2, ...S_k\}$ and will refer to the total number of subscriptions maintained as $|\mathcal{S}|$. Each subscription $S_i \in \mathcal{S}$ that corresponds to a set of terms denoted as $S_i = \{t_1, t_2, ..., t_n\}$ is defined as $S_i \subseteq \mathcal{V}_\mathcal{S}$. The size of S_i denoted as $|S_i|$ corrsponds to the number of terms it contains. Publishers publish content to the system in the form of news items which are also composed of terms. We will use $\mathcal{I} = [I_1, I_2, ...I_n]$ to denote the stream of published news items. For every item $I_j \in \mathcal{I}$ it holds that $I_j \subseteq \mathcal{V}_\mathcal{S}$. The size of I_k such that $I_k = \{t_1, t_2, ...t_m\}$, corresponds to the number of terms of I_k and is denoted as $|\mathcal{I}_k|$.

Definition 1 (Covering Property)**.** *An item I_k is said to* fully *cover a subscription S_i ($I_k \succeq S_i$) if and only if $\forall t_j \in S_i \rightarrow t_j \in I_k$.*

Definition 2 (Matching Semantics - Broad match)**.** *We say that a subscription S_i matches an item I_k if and only if $I_k \succeq S_i$.*

Lemma 1. *Given a subscription S_i and an item I_k then $I_k \succeq S_i \rightarrow |I_k \cap S_i| = |S_i|$.*

Problem

Given an item I_k, we want to find the set of subscriptions $\mathcal{S}_M \subseteq \mathcal{S}$ for which a match occurs (w.r.t. Definition 2).

Example 2.1

To better illustrate how matching is applied, suppose the motivating example presented in Table 2.1. Matching item $I_1 = \{t_{12}, t_1\}$ against the set of subscriptions, will result to the set of matched subscriptions $\mathcal{S}_\mathcal{M} = \{S_4\}$. Subscription S_4, is in the resulting answer because both terms t_1 and t_{12} are also contained within item I_2; it holds that $S_4 \subset I_1$. To demonstrate the differences from retrospective search queries consider example $I_2 = \{t_{24}\}$. Apparently, for I_2 it holds that $I_2 \subset S_2$. However, the query answer would be $\mathcal{S}_\mathcal{M} = \{\}$ since according to Definition 2, it has to hold that $I_2 \supseteq S_2$ and not vice versa. It is clear that the opposite containment relations with regard to what is being queried in each case need to hold. □

Table 2.1: Example of a set of keyword based subscriptions

Subscription	Terms
S_1	$(t_1 \wedge t_2 \wedge t_4)$
S_2	$(t_1 \wedge t_{24})$
S_3	$(t_1 \wedge t_2 \wedge t_3)$
S_4	$(t_1 \wedge t_{12})$
S_5	$(t_2 \wedge t_4)$
S_6	$(t_2 \wedge t_3 \wedge t_{13})$

Despite the fact that we only consider conjunctive (AND) semantics, other operators under the boolean model such as negation (NOT) and disjunction (OR) can also be applied, by pre converting such complex subscriptions to their equivalent DNF form and treating each atomic conjunctive formula as a normal subscription.

2.1 Naive - Brute Force Method

Assume that no index is built over the set of subscriptions. The simplest and most straightforward way to match a news item I_k against a set of subscriptions $\mathcal{S}$ is to iterate over $\mathcal{S}$ and for each term t_j contained within a subscription $S_i \in \mathcal{S}$ check if that particular term t_j is also present in the item I_k. As soon as we encounter a term t_j of a subscription S_i that is not also contained within the item under test, we proceed to the next subscription S_{i+1}. If all terms t_i of S_i are present in I_k, we report a match. This naive approach is illustrated in Algorithm 1 and will here after be referred to as the brute-force method.

Algorithm 1: $BF\_MATCH(I_k, \mathcal{S})$

Require: An item I_k for which $|I_k| \geq 0$.
Require: The set of submitted subscriptions $\mathcal{S}$.

```
1:  S_MATCHED ← {}
2:  for all subscriptions S_i ∈ S do
3:      no_match ← false
4:      for all terms t_j ∈ S_i do
5:          if t_j ∉ |I_k| then
6:              no_match ← true;
7:              break;
8:          end if
9:      end for
10:     if no_match = false then
11:         S_MATCHED ← S_MATCHED ∪ {S_i}
12:     end if
13: end for
14: return S_MATCHED
```

The brute force method does not require any additional space other than the subscriptions themselves. The space requirements are $O(|\mathcal{S}| * |\mathcal{S}_i|_{AVG})$. Because set membership between each term t_j of every subscription S_i and the item (line 5 of Algorithm 1) is performed several times

(critical inner loop), methods for efficiently testing set membership need to be employed. One way to efficiently perform such a test is to build a hash table for the incoming item in order to quickly determine if a particular term is present within that specific item or not. The lookup of a term in the item using a hash table can be done in $O(1)$. Therefore, the time complexity is equal to $O(|\mathcal{S}| * |\mathcal{S}_i|_{AVG})$.

Even if testing for set membership is performed in a timely manner, the number of such tests that need to be performed when the brute-force method is considered is still large. Recall from the previous chapter that we anticipate $|\mathcal{S}| \approx 10^6$ and $|S_i| \approx 3$. Such figures, render the brute force method inappropriate given the high publishing rate of news items. Obviously there exists a dependency in $|\mathcal{S}|$ which is undesirable.

One way for improving matching performance when employing the brute-force method is by exploiting statistical information (if any) regarding the probability of a term's presence(absence) within a subscription. If the vocabulary V_S of subscription terms provides information about the frequency of a particular term then, by ranking the terms according to that frequency and storing them in reverse frequency rank order would result to better matching performance [45]. The idea is to exploit the conjunctive semantics as applied to matching (see Definition 2). Recall, that the brute-force method stops testing a particular subscription S_i as soon as it encounters a term t_j of S_i that is not also contained in the item I_k being evaluated. Hence, by ordering the terms of a subscription from the least frequent to the most frequent would ensure a faster matching since non matching subscriptions will be discarded as early as possible (only few terms per subscription need to be tested in order to decide a match). Consider example 2.2 bellow.

Example 2.2

Consider the set of subscriptions of our motivating example in Table 2.1 and assume that the term subscripts denote their corresponding frequency ranks; we consider t_1 to be more frequent than t_2 which in turn is more frequent that term t_3 and so on. Suppose now that we want to find subscriptions matching item $I_1 = \{t_1, t_2, t_3\}$. The brute force method starts by testing the first subscription, S_1. The least frequent term which is t_4 is first considered. Since t_4 is not present within I_1 the algorithm proceeds to the next subscription,S_2. The algorithm next processes term $t_{24} \in S_2$, verifies that it is not also within I_1 and continuous to subscription S_3. This procedure is repeated for all stored subscriptions S_i. The total number of set membership tests for item I_1 in the improved brute force method is equal to 8 as opposed to the naive case where 10 tests would need to be performed. Observe that this frequency based improvement does not apply for subscription S_3 (which is reported as matched) where all three terms need to be examined. □

The achieved performance gain for the optimized approach depends on the number of frequent and infrequent terms the subscriptions contain. If subscriptions contain at least one infrequent term but many frequent terms then frequency based term matching would result to a significant gain. It must be stressed here though, that this simple optimization will not result to any gain for subscriptions belonging to the resulting set of matched subscriptions, since by definition all of their containing terms will be tested for set membership against I_k.

2.2 Count based Subscription Index

The brute-force algorithm needs to examine all of the subscriptions before returning the set of matched subscriptions. Considering the fact that within our setting we need to be able to handle millions of subscriptions it is obvious that such a simple solution does not scale. Any efficient

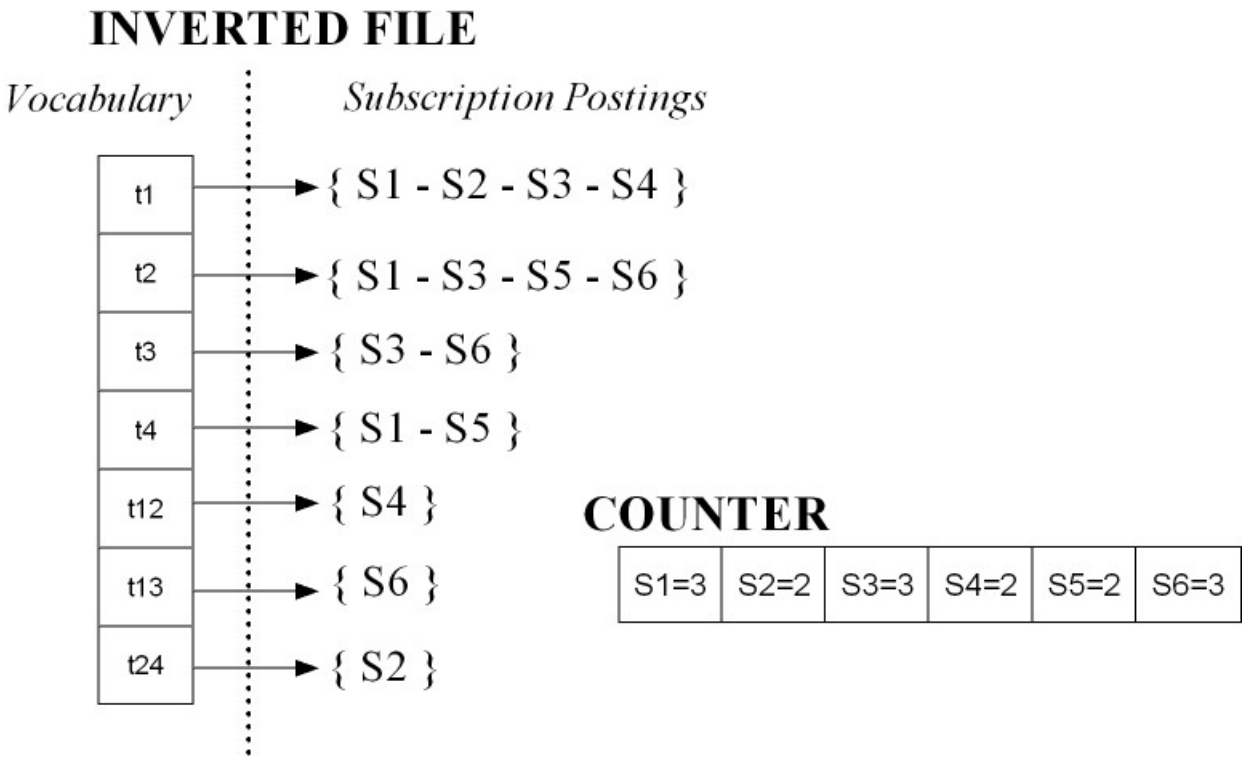


Figure 2.2: Count based index built over motivating example presented in table 2.1

solution to the matching problem should eliminate potentially non matching subscriptions as soon as possible. This early subscription pruning can be achieved if make use of the following observation: according to our matching semantics (see Definition 2), every subscription term must also be present in the news item under consideration; a subscription containing terms not belonging to a particular news item can safely be excluded from the resulting set of matched subscriptions. More formally, we can reject subscriptions that contain any term $t_k \in V_S$ for which it holds, $t_k \notin terms(I)$. The idea is to maintain an inverse mapping from terms to subscriptions that confines the original search space to subscriptions only containing a term which is also present within the news item being matched.

In a nutshell, the inverted file data structure can be used to map every term $t_j \in \mathcal{V}_S$ to the set of subscriptions that contain that specific term since it provides fast access methods to subscriptions containing a specific term. The inverted file structure is composed of two main parts namely, the vocabulary of terms and the inverted sets(lists) of subscriptions. Each vocabulary entry stores the term itself $\forall t_j \in \mathcal{V}_S$ and a pointer to the inverted set of subscriptions containing t_j. We will use the notation $Postings(t_j)$ here after to denote the set of subscription postings assigned to term t_j.

Figure 2.2 illustrates the inverted file built over the set of subscriptions of Table 2.1. Consider for example subscription $S_6 = \{t_3, t_{13}\}$. A posting entry labelled with S_6 is contained in both the inverted sets corresponding to each of the subscription terms t_3 and t_{13}. Observe that the inverted set of postings for term t_2 is equal to $Postings(t_2) = \{S_1, S_3, S_5, S_6\}$ and is of size $|Postings(t_2)| = 4$. Apparently, the size of a particular posting list is equal to the frequency of a term in the total set of subscriptions $\mathcal{S}$. Thus, we anticipate that the structure of the inverted file to reflect the distribution that the terms of the vocabulary $\mathcal{V}_S$ obey.

When processing a boolean query against an inverted file structure of documents, we can

simply get the answer by applying set operations on the inverted sets obtained for each particular term[6]. For example AND (i.e. conjunctive semantics) operators are processed by intersecting the inverted sets; OR (disjunctive semantics) operators are processed by unioning(merging) the obtained inverted sets. However, this technique cannot be used in our context. Despite the fact that we also assume a conjunctive subscription model, simply intersecting the inverted sets of the terms does not lead to valid results. To better illustrate this fact, consider the example of Figure 2.2 and suppose we want to match news item $I_1 = \{t_1, t_{12}, t_{24}\}$. The answer set when simply intersecting the inverted sets of terms of I_1 (i.e. $Postings(t_1) \cap Postings(t_{12}) \cap Postings(t_{24})$) would be equal to the empty set $\{\}$; no matching subscriptions will be returned. This however is not valid, since for I_1 we should obtain $S_{MATCHED} = \{S_2, S_4\}$.

Instead of intersecting the obtained inverted sets for given terms t_j of a news item I_k, we can alternatively union them. In our motivating example, unioning the inverted sets of terms for $I_1 = \{t_1, t_{12}, t_{24}\}$ would result to the subscription set $\{S_1, S_2, S_3, S_4\}$. Once again this is not equal to the expected $S_{MATCHED} = \{S_2, S_4\}$. However, it holds that the unioned set of subscriptions is a superset of the actual answer. This superset relation can be explained by the following observation: every subscription in the unioned set contains at least one term t_j such that t_j is also contained within news item I_1. It holds that: $\forall S_i \in S_{UNIONED} \rightarrow |S_i \cap I_1| \geq 1$. Along with the correct subset of subscriptions that will potentially get matched, false positives are also obtained. These false positives refer to subscriptions that are *partially* but not *fully* covered by news item I_1 (Lemma 1). To obtain the correct set of matched subscriptions, further processing is needed for excluding such false positive subscriptions. To determine if the subscription $S_i \in S_{UNIONED}$ is an actual answer one would need to further check if all of its terms $t_j \in S_i$ are also contained within the news item. This procedure referred to here after as *IF_Match* is presented in Algorithm 2.

Algorithm 2: $IF\_MATCH(I_k, \mathcal{IF}_{\mathcal{S}})$

Require: An news item I_k for which $|I_k| \geq 0$.
Require: The inverted file index, $\mathcal{IF}_{\mathcal{S}}$ built over the set of subscriptions $\mathcal{S}$

1: $\mathcal{S}_{UNIONED} \leftarrow \{\}$
2: **for all** $t_j \in I_k$ **do**
3: $\quad \mathcal{P}_{t_j} \leftarrow$ obtain $Postings(t_j)$ from $\mathcal{IF}_{\mathcal{S}}$
4: $\quad \mathcal{S}_{UNIONED} \leftarrow \mathcal{S}_{UNIONED} \cup \mathcal{P}_{t_j}$
5: **end for**
6: $\mathcal{S}_{MATCHED} \leftarrow BF\_MATCH(I_k, \mathcal{S}_{\mathcal{UNIONED}})$
7: **return** $\mathcal{S}_{MATCHED}$

The *IF_Match* algorithm is a two step procedure; the first step involves the union operation where all inverted sets are traversed and their subscriptions are merged (lines 2-5 of Algorithm 2); the second step involves the traversal of all unioned subscriptions from step 1, to exclude false positives (line 6 of Algorithm 2). The greater the discriminative power the terms of a news item have, the greater the gain of using the inverted file. If a news item contains terms that do not appear in many subscriptions (i.e. are discriminative) then obtaining the inverted sets of terms t_j, unioning them, and then performing a brute force matching upon them will result to an efficiency gain when compared to the naive method. This gain is due to the pruning of the subscription search space incurred by step 1 *IF_Match*. To better illustrate this, recall our motivating example of Figure 2.2 and suppose we want to match the set of subscriptions against news item $I_2 = \{t_2, t_4\}$. The brute-force method would require to scan through the

whole set of subscriptions $\mathcal{S}$, where as the *IF_Match* would only need to examine subscriptions $\{S_1, S_3, S_5, S_6\}$ corresponding to $Postings(t_2) \cup Postings(t_4)$.

Although the *IF_Match* reduces the search space of subscriptions it still requires two passes over the indexed subscriptions; one pass to determine the unioned subscription set; a second one to eliminate any possible false positives within the obtained set. Additionally, further subscription pruning performed on the candidate set produced in step 1 additionally requires, either supplementary data structures or accessing the subscriptions themselves. An improvement could be achieved based upon the following observation: when iterating over the inverted sets for the terms t_j of I_k, a particular subscription posting S_i could be considered more than one times. The number of times S_i is considered is equal to the common terms S_i and I_k share; i.e. $|S_i \cap I_k|$. Recall that, according to Lemma 1, for a news item I_k to match a subscription S_i it has to hold that $|S_i \cap I_k| = |S_i|$. An extension to the *IF_Match* method referred to as counting method (count based index) that exploits this observation is presented in the next subsection. The idea is to additionally maintain a counter per subscription in order to further improve efficiency by matching in a single pass over the set of subscription postings. Along with the inverted file structure, this technique also considers a *Count*[3] structure that facilitates matching. As we can see in Figure 2.2 the Count(er) maps every subscription to the total number of terms it contains and is used to keep track of the number of matching terms per subscription in order to verify satisfiability according to Lemma 1 when evaluating a news item.

2.2.1 Count Based Index - Construction

When adding a subscription S_i into the index, an entry labelled S_i is added, to the postings list associated to every term t_i of S_i. Additionally, an entry is inserted in the *Counter* map with the total number of terms, $|S_i|$, subscription S_i contains. Algorithm 3 illustrates the procedure.

Algorithm 3: $CB\_ADD(S_i)$

Require: A subscription S_i with for which $|S_i| \geq 0$.

1: $cnt \leftarrow 0$
2: $PostSet \leftarrow \{\}$
3: **for all** terms t_j in S_i **do**
4: $\quad PostSet \leftarrow$ *get subscription postings set for term* t_j
5: $\quad PostSet \leftarrow PostSet \cup \{S_i\}$
6: $\quad cnt \leftarrow cnt + 1$
7: **end for**
8: $Counter[S_i] \leftarrow cnt$

Example 2.3

Consider that we want to index subscription $S_3 = \{t_1, t_2, t_3\}$. Initially, the postings set for term $t_1 \in S_3$ is obtained and an entry labelled with S_3 is added to it. This is repeated for both other terms t_2 and t_3 of subscription S_3. The number of additions equals to the size $|S_3| = 3$. Additionally, an entry within the *Counter* map is inserted for S_3 with a value equal to 3 as depict in Figure 2.2. □

[3] will be referred to simply as Counter

The set of parameters that affect construction time, memory requirements and matching time of the subscription indices are summarized in Table 2.2. Parameters have been categorized into two distinct classes, W(workload) and R(representation), respectively [8]. The first class of parameters refer to characteristics that are machine independent and describe the workload under consideration, such as the total number of subscriptions, the vocabulary size, etc. The latter refer to machine dependant characteristics which capture internal representation and influence internal subscription storage. This set of parameters as defined in table 2.2 will be used throughout this thesis.

Table 2.2: Parameters that characterize the workload

Parameter	Description	Category
$\|\mathcal{S}\|$	Total Number Of Subscriptions	W
$\|V_S\|$	Vocabulary Size	W
$\|S_i\|_{AVG}$	Expected Subscription Size	W
$\|I_i\|_{AVG}$	Expected news item Size	W
$w(c)$	Width of Counter Entry	R
$w(v)$	Width of Vocabulary Entry	R
$w(p)$	Width of Subscription Posting Entry	R

Construction Time Requirements

Adding a subscription S_i requires for every term $t_j \in S_i$, one vocabulary probe to obtain the respective inverted set as well as an addition to the corresponding posting set. One counter update should also be performed. Hence, the total number of operations required to insert a subscription S_i is equal to:

$$\begin{aligned} Time(ADD\_SUBSCRIPTION) &= |S_i|_{AVG} * Time(Vocabulary\_Probe) + \\ &\quad |S_i|_{AVG} * Time(Set\_Addition) + \\ &\quad Time(Counter\_Insertion) \end{aligned} \tag{2.1}$$

Assuming an inverted set addition, and a *Counter* update can be performed in O(1) we can deduce that the time needed to insert a particular subscription $S_i \in |\mathcal{S}|$ into the Count based index can be performed in $O(|S_i|_{AVG})$. It is obvious that adding a subscription to the index is scalable as it not affected by the total number of indexed subscriptions $|\mathcal{S}|$

Memory Requirements

In the following we will present an analysis of the memory usage of the count based subscription index. Recall that the count based approach is composed of two main structures, the *inverted file* and the *Counter*. The *inverted file* is further decomposed to the *vocabulary* which stores the set of terms found within $\mathcal{S}$ and the *subscription postings* which store the set of subscription corresponding to particular terms. Therefore, the overall memory required by the index is:

$$\begin{aligned} Size(Index) &= Size(IF) + Size(Counter) \\ &= Size(Vocbualry) + Size(Postings) + Size(Counter) \end{aligned} \tag{2.2}$$

Each subscription indexed has a corresponding entry in the *counter*. Since every entry occupies $w(c)$ space, the memory required by the *Counter* is equal to:

$$Size(Counter) = w(c) * |\mathcal{S}| \tag{2.3}$$

Let $Pr(t_j)$, denote the probability assigned to term t_j. $Pr(t_j)$ corresponds to the observed frequency of t_j normalized over the size of $\mathcal{V}_S$. Hereafter we assume that the probability of appearance of a term $t_j \in S_i$ is independent of every other term $t_k \in S_i$. In addition we assume that all subscriptions have the same size and use the notation $|S_i|_{AVG}$ to denote the it. In a similar manner we assume a constant news item size denoted as $|I_k|_{AVG}$.

A subscription S_i of size $|S_i|$ does not contain term t_j if and only if the 1^{st}, the 2^{nd}, ... , the $|S_i|^{th}_{AVG}$ term of S_i is not t_j. Hence, the probability that t_j appears within a subscription S_i with respect to $Pr(t_j)$, is:

$$\begin{aligned} Pr(t_j \in S_i) &= 1 - Pr(t_j \notin S_i) \\ &= 1 - (1 - Pr(t_j))^{|S_i|_{AVG}} \end{aligned} \tag{2.4}$$

Clearly, in order for a term t_j to not be present within the vocabulary $\mathcal{V}_\mathcal{S}$ it has to hold that t_j does not appear in any of the subscriptions S_1,S_2,...,$S_{|\mathcal{S}|}$ of $\mathcal{S}$. Thus, the probalitity $Pr(t_j \in \mathcal{S})$ is equal to:

$$\begin{aligned} Pr(t_j \in \mathcal{S}) &= 1 - Pr(t_j \notin \mathcal{S}) \\ &= 1 - \prod_{i=0}^{|\mathcal{S}|}(1 - Pr(t_j \in S_i)) \\ &\overset{(2.4)}{=} 1 - \left(1 - (1 - Pr(t_i))^{|S_i|_{AVG}}\right)^{|\mathcal{S}|} \end{aligned} \tag{2.5}$$

Having calculated the probability of a particular term t_j to be stored within the vocabulary, the vocabulary size could be defined as the expected number of stored terms, multiplied by the space occupied by a vocabulary entry($w(v)$):

$$\begin{aligned} Size(Vocabulary) &= \left[\sum_{j=0}^{|V_S|} Pr(t_j \in \mathcal{S})\right] * w(v) \\ &\overset{(2.5)}{=} \left[\sum_{j=0}^{|V_S|} 1 - \left(1 - (1 - Pr(t_j))^{|S_i|_{AVG}}\right)^{|\mathcal{S}|}\right] * w(v) \end{aligned} \tag{2.6}$$

Given a term t_j, a subscription posting is inserted into $Postings(t_j)$, every time t_j is encountered within a subscription $S_i \in \mathcal{S}$. The size of each inverted set $Postings(t_j)$ is thus equal to the number of times t_j is found within $|\mathcal{S}|$. Let random variable, $P^{t_j}_{size}$, express the expected number of subscription postings a particular term $t_j \in V_S$ contains. $P^{t_j}_{size}$, follows the binomial distribution $Bin\left[|\mathcal{S}|, Pr(t_j \in S_i)\right]$.

Assuming that the mean of the binomial distribution $Bin\left[x; N, Pr\right]$ is equal to $N * Pr$, then using $E\left[P^{t_j}_{size}\right]$ to denote the average number of postings for term t_j, we can define the total

space required for storing the set of postings as:

$$
\begin{aligned}
Size(Postings) &= \left[\sum_{j=0}^{|V_{S|}} (E\left[P_{length}^{t_j}\right])\right] * w(p) \\
&= \left[\sum_{j=0}^{|V_{S|}} (|\mathcal{S}| * Pr(t_j \in S_i)\right] * w(p) \\
&\overset{(2.4)}{=} \left[\sum_{j=0}^{|V_{S|}} (|\mathcal{S}| * (1-(1-Pr(t_j))^{|S_i|_{AVG}})\right] * w(p) \qquad (2.7)
\end{aligned}
$$

The space consumed by the index in total is:

$$
\begin{aligned}
Size(index) \overset{(2.3),(2.6),(2.7)}{=}\ & |\mathcal{S}| * w(p) + \\
& \left[\sum_{j=0}^{|V_{S|}} 1-(1-(1-Pr(t_j))^{|S_i|_{AVG}})^{|\mathcal{S}|}\right] * w(v) + \\
& \left[\sum_{j=0}^{|V_{S|}} (|\overline{S}| * (1-(1-Pr(t_j))^{|S_i|_{AVG}})\right] * w(p) \qquad (2.8)
\end{aligned}
$$

2.2.2 Count Based Index - Matching

The Count based subscription index uses a copy of the *Counter* to keep track of the number of terms currently satisfied per subscription and verify for matched subscriptions. The procedure is illustrated in Algorithm 4 below. Every time a new matching process is launched, the *Counter_cpy* is initialized as an exact copy of the *Counter*. When matching a news item I_k against the set of subscription currently indexed, the Count based approach iterates through every term $t_j \in I_k$ and obtaines the corresponding inverted set of postings (line 3). Next, for each subscription posting S_i within the inverted set, a lookup in the *Counter_cpy* is performed, and the corresponding subscription value is decremented by one (line 5). If the decremented value of a subscription S_i under test is equal to zero, the S_i is returned as matched.

Algorithm 4: $CB\_MATCH(I_k)$

Require: An news item, I_k for which $|I_k| \geq 0$.

```
1:  Counter_cpy ← copy of Counter
2:  for all terms t_j ∈ I_k do
3:      PostSet ← Posting(t_j)
4:      for all S_i in PostSet do
5:          Counter_cpy[S_i] ← Counter_cpy[S_i] − 1
6:          if Counter_cpy[S_i] = 0 then
7:              notify subscriber S_i
8:          end if
9:      end for
10: end for
```

Example 2.4

To better demonstrate how matching is performed consider news item $I_1 = \{t_1, t_{24}, t_{12}\}$ evaluated over the set of subscriptions of our motivating example defined in Table 2.1. Initially, $Counter\_cpy = \{S_1 : 3, S_2 : 2, S_3 : 3, S_4 : 2, S_5 : 2, S_6 : 3\}$. After processing the first term $t_1 \in I_1$, the *Counter_cpy* will be update to *Counter_cpy* $= \{\mathbf{S_1 : 2}, \mathbf{S_2 : 1}, \mathbf{S_3 : 2}, \mathbf{S_4 : 1}, S_5 : 2, S_6 : 3\}$ since S_1, S_2, S_3 and S_4 are all contained in $Postings(t_1)$. Processing the second term $t_{24} \in I_1$ would further update the *Counter_cpy* to $\{S_1 : 2, \mathbf{S_2 : 0}, S_3 : 2, S_4 : 1, S_5 : 2, S_6 : 3\}$ and result to a reported match for subscription S_2 since its corresponding value in the *Counter_cpy* map after decrementing, is equal to zero. In a similar manner, processing the final term $t_{12} \in I_1$ would result to *Counter_cpy* $= \{S_1 : 2, S_2 : 0, S_3 : 2, \mathbf{S_4 : 0}, S_5 : 2, S_6 : 3\}$ and a reported match for subscription S_4. □

Instead of copying the whole Counter, an alternate approach could consider counting upwards by copying only the values of subscriptions considered when traversing the postings sets. To determine the set of matching subscriptions however, this approach would require an additional pass over the copied values in order to verify equality between the number of times a subscriptions was considered and the number of tems it contains.

Matching Time Requirements

The time complexity of matching an item I_k against the set of indexed subscriptions $\mathcal{S}$ is equal to the number of times the critical inner loop (line 4-9 of algorithm 4) is executed. This is equal to the sum of the sizes of all of the inverted sets corresponding to terms t_j within item I_k. Using the same notation $E\left[P^{t_j}_{length}\right]$ as above to express the average expected size of the inverted set of a particular term t_j and assuming that the time needed to perform a counter decrement and test is equal to $time(Counter\_Decr)$ we can define the time needed to perform matching as:

$$\begin{aligned} Time(Match) &= \left[\sum_{j=0}^{|I_k|}(E\left[P^{t_j}_{length}\right])\right] * time(Counter\_Decr) \\ &= \left[\sum_{j=0}^{|I_k|}(|\mathcal{S}| * Pr(t_j \in S_i)\right] * time(Counter\_Decr) \\ &\overset{(2.4)}{=} \left[\sum_{j=0}^{|I_k|}(|\mathcal{S}| * (1 - (1 - Pr(t_i))^{|S_i|_{AVG}})\right] * time(Counter\_Decr) \quad (2.9) \end{aligned}$$

2.2.3 Count Based Index Remarks

Unlike the brute-force approach, the Count based index partitions the set of subscriptions according to the terms each subscription contains. It thus confines the search space upon evaluation resulting to more efficient matching. Table 2.3 summarizes the *worst-case* performance of the naive and Count based techniques (see Algorithms 1 and 3-4) with regard to their memory requirements and time (construction and matching).

As opposed to the IF_Match that only uses the inverted file structure for matching, the count base approach does not require any supplementary data structure nor accessing the subscriptions themselves in order to return the set of matched subscriptions. A single pass over the inverted set of subscriptions along with counting is sufficient to decide the satisfiability of the matching

Table 2.3: Performance comparison(worst case) of Brute-Force and Count-Based methods

	Construction Time	Construction Memory	Matching Time
Brute-Force	$O(\|S_i\|_{AVG})$	$O(\|\mathcal{S}\| * \|S_i\|_{AVG})$	$O(\|S_i\|_{AVG} * \|\mathcal{S}\|)$
Counting	$O(\|S_i\|_{AVG})$	$O(\|\mathcal{S}\| * (1 + \|S_i\|_{AVG}) + \|\mathcal{V}_S\|)$	$O(\|S_i\|_{AVG} * \|\mathcal{S}\|)$

conditions. This however, implies that subscription are *redundantly* stored within the inverted file. Recall, that the number of times a particular subscription is stored in different term postings sets is equal to its size (number of terms it contains).

For the Count-based index, matching is affected by the size of the news item and the expected size of the subscription postings corresponding to the terms the matching news item contains. Although the inverted index does reduce the initial search space of subscriptions, the number of candidate[4] subscriptions could still remain large if the term distribution, for both subscriptions and news items, is skewed. One such case, which is realistic, is when the terms of the subscriptions obey the long tale (Zipf) distribution. If this is the case then a great fraction of the terms are associated to a few subscriptions and only a small number of terms are associated to many subscriptions. To demonstrate how this applies, consider as an example the frequent term 'greece'. Since, 'greece' appears in a great number of subscriptions, it is expected that it's corresponding inverted set to be also large. Apparently, when matching news items that also contain the term 'greece', a large number of subscription postings would need to be traversed and hence matching becomes costly. The count based approach that utilizes the inverted file structure can be considered wasteful under the afore mentioned circumstances since, a great fraction of the candidate subscriptions initially considered will not result to a match as they would also contain additional terms not contained within the news item under consideration.

This problem is not new and has been extensively studied within the IR community. Several techniques aiming at reducing the retrieval cost, impacted by retrieval and decoding of the large inverted lists, have been devised [48]. One optimization that exploits the best match semantics as applied to text search engines, considers partial inverted list evaluation. More specifically, inverted lists are sorted by decreasing similarity impact and processing stops as soon as a given threshold is exceeded [44]. However, such techniques can not be applied within the specific context. The matching semantics (see Definition 2), imply a universal qualifier on the terms of a subscription (all of the terms of the subscription must be present in the news item) which in turn does not permit partial inverted list evaluation. Hence, solutions that assume a more fine grained partitioning of the subscription search space need to be devised.

2.3 Trie Subscription Index

The Count based index, that utilizes the inverted file structure, performs a one level partitioning by grouping the set of subscriptions according to the terms they contain. However, as discussed in the previous section, if both news items and subscriptions follow the same skewed distribution then matching could become inefficient as a large number of subscription postings would need to be traversed. In addition, the count index requires to maintain an additional structure for counting. We overcome both of these limitations by devising a tree based approach for storing subscriptions. As opposed to the flat vocabulary of the inverted file, organizing the set of terms within the vocabulary into a tree hierarchy leads to a more fine grained partitioning

[4]the term candidate subscription refers to subscriptions which are considered when matching (a Counter decrement has been perfromed)

of the subscription search space and thus enables a more efficient evaluation. Moreover, the hierarchical structure alleviates the need for maintaining an additional structure counting as counting would be performed implicitly.

We use a tree structure built over the vocabulary of subscriptions that reflects *subscription covering* relations. In accordance to Lemma 1, a subscription S_i is said to fully cover a subscription S_j, denoted as $S_i \succeq S_j$, if and only if $\forall t_j \in S_j \rightarrow t_j \in S_i$. Each node of the tree contains a specific term $t_j \in \mathcal{V}_S$, a list of pointers to child nodes and the set of subscriptions assigned to that particular node. A subscription stored at node n of the tree is assigned to the set of terms that corresponds to the union of all terms found in the path from the root to node n. Edges identify covering relations. For example, if a subscription S_j is stored at a particular node n, then for every other subscription S_i stored at a descendant node of n it holds that $S_i \succ S_j$. Subscriptions belong to the same node if and only if they are identical (they contain exactly the same set of terms). To better demonstrate the idea consider the Example 2.5 following.

Example 2.5

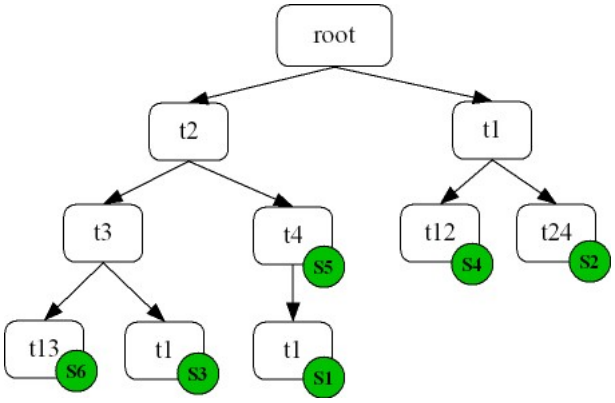
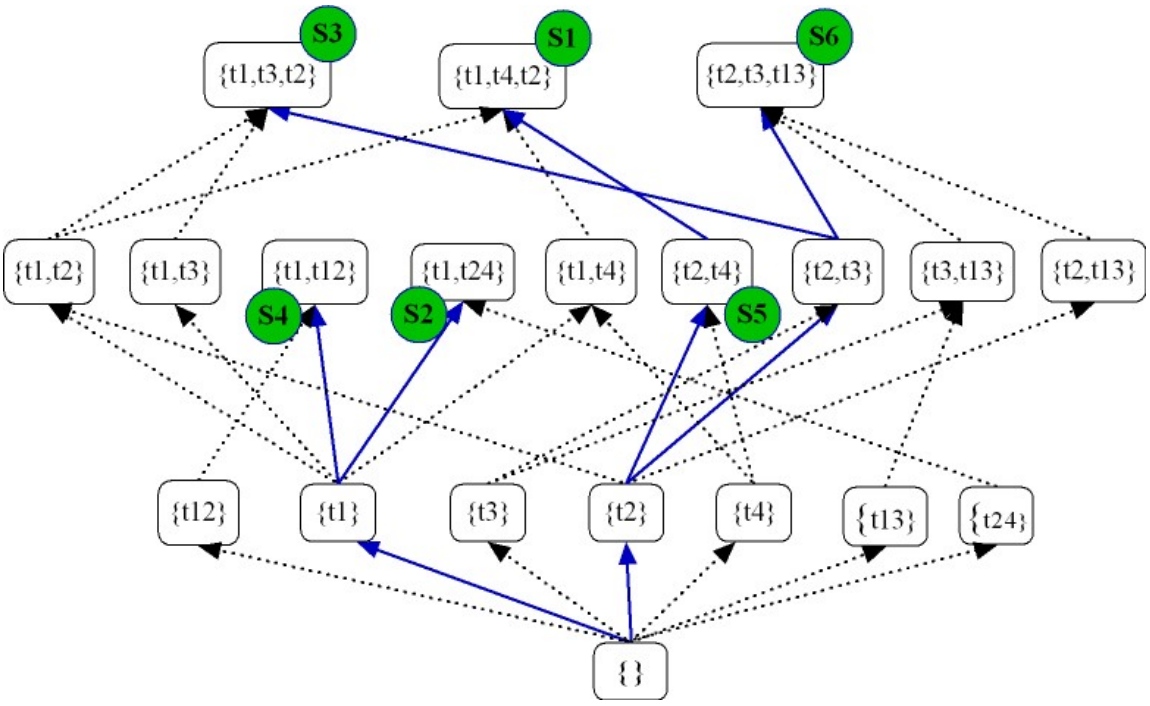


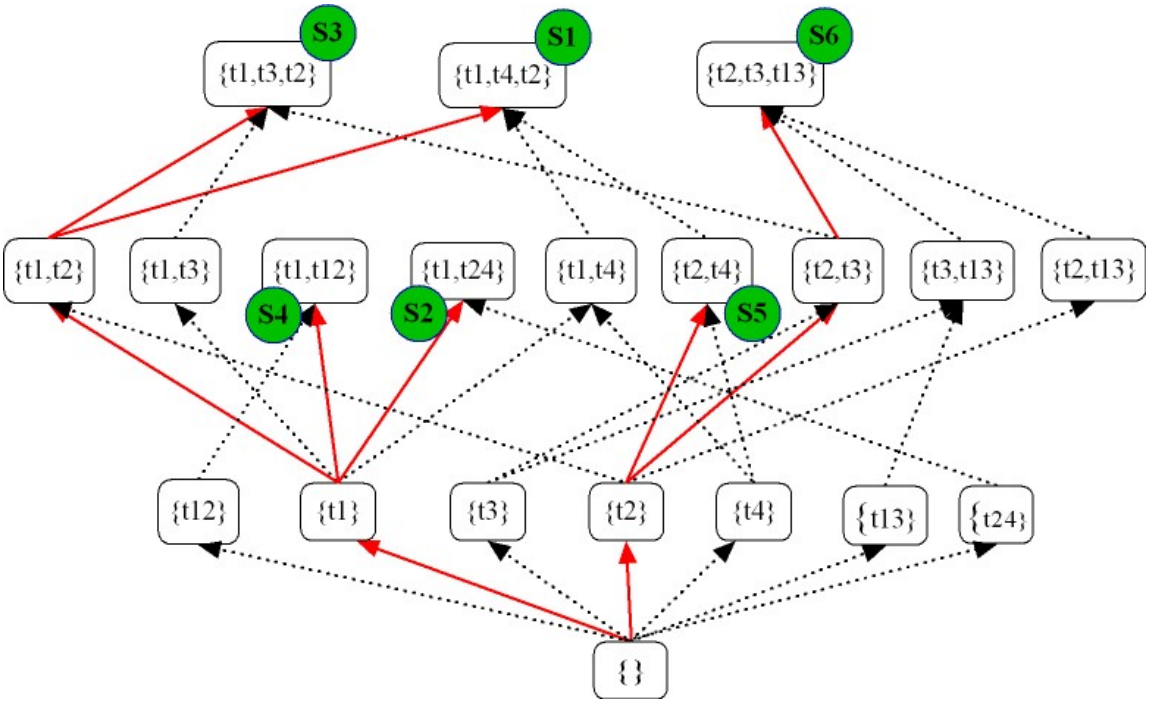
Figure 2.3: Tree based index built over the subscriptions of Table 2.1.

Figure 2.3 depicts a possible tree structure build over the set of subscriptions of Table 2.1. Consider subscription $S_3 = \{t_1, t_2, t_3\}$ and let us define the node at which S_3 is stored as n_3. Observe that the set of terms assigned to S_3 is equal to the union of the terms found in the path form the root to the node n_3. In a similar manner, subscription S_5 corresponds to the set of terms $S_5 = \{t_4\} \cup \{t_2\}$. Subscription S_1 fully covers subscription S_5 since it is stored at a descendant node of that to which S_5 is stored. The common subset that both S_5 and S_6 share is equal to the path from the root the sub-tree both the subscriptions belong to. For this specific example the common subset (i.e. prefix) shared by both subscriptions S_5 and S_6 is equal to $\{t_2\}$. □

The example of Figure 2.3 is only one of the possible tree structures built over the set of subscriptions of Table 2.1. Depending on the covering relations considered, a different tree structure could be constructed in each case. If we consider the lattice of all of the partially ordered inclusion (covering) relations for the set of vocabulary terms then, the number of different trees indices that could be constructed is equal to all of the possible traversals. Figure 2.4 depicts two possible cases. Observe that Figure 2.4(a) corresponds to the tree index presented previously in Example 2.5. Clearly, tree index construction is non deterministic.



(a) A Traversal of covering relations corresponding to the Trie index of Figure 2.3



(b) Another possible traversal of covering relations corresponding to Trie index of Figure 2.5

Figure 2.4: The partial ordered set of inclusion(covering) relations for the vocabulary of our motivating example of Table 2.1

In order to overcome the issue of non determinism, additional information needs to be considered with regard to construction. We thus consider a *Trie* structure for indexing subscriptions by enforcing total order on the terms of the vocabulary. This total order could either be random, or follow for example the term frequency ranking in subscription/news items if available. Using the latter approach leads to several interesting findings which are discussed in later sections of this work.

2.3.1 Trie Index - Construction

In the context of web syndication, we consider subscriptions as set of words (terms). Although such a model is suitable for the inverted file index of the count based approach, it cannot be applied for the Trie index. The Trie data structure is an ordered tree data structure and thus subscriptions should be modelled as sequences of words. A total ordering among the terms of the vocabulary $\mathcal{V}_S$ needs to be applied. Let $\mathcal{ORDER} : \mathcal{V} \mapsto \mathbb{R}$ denote the mapping that maps a particular term t_j to its corresponding order within the vocabulary. In the rest of our thesis, unless specified differently, term subscripts will be used to denote ordering. The $\mathcal{ORDER}$ function which is assumed to be known before hand is bijective since for any two terms $t_j, t_k \in \mathcal{V}_S$ for which $j \neq k$ it has to hold that $\mathcal{ORDER}(t_j) \neq \mathcal{ORDER}(t_k)$; different terms should never map to the same rank order.

Subscription addition for the Trie index is illustrated in Algorithm 5 below. When adding a new subscription S_i to the Trie index the algorithm performs a top down traversal starting from the index root. Initially, the terms of S_i are sorted according to their ranking order (line 1). At the first step of the algorithm, the path corresponding to the first term of the ordered set of terms is followed. This procedure is repeated for every term $t_j \in S_i$. If a particular path does not exist, then a new node labelled with the term under consideration is created and inserted into the Trie structure (lines 5-8). The node at which the top down traversal, after consuming the whole set of terms concludes, is where S_i is finally stored. Figure 2.5 depicts the Trie index built for the set of subscriptions of Table 2.1.

Algorithm 5: $TRIE\_ADD(S_i)$

Require: A subscription, S_i with $|S_i| \geq 0$.

```
1:  sorted_terms ← sort(S_i)
2:  currNode ← root
3:  for all terms t_j in sorted_terms do
4:      childNode ← get child of currNode corresponding to t_j
5:      if childNode = null then
6:          childNode ← new node with label t_j
7:          set childNode as child of currNode
8:      end if
9:      currNode ← childNode
10: end for
11: add S_i to currNode
```

Example 2.6

Consider in Figure 2.5 that all subscriptions except $S_6 = \{t_2, t_1 3, t_3\}$ have already been added into the index. The $\mathcal{ORDER}$ function is assumed to be known. Recall that term subscripts denote the

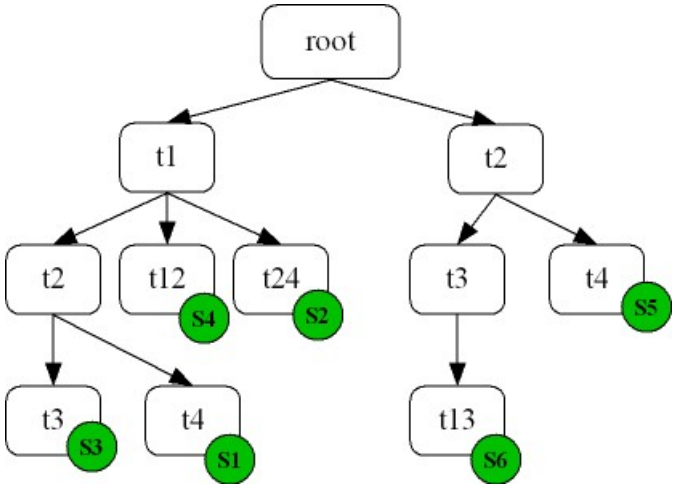


Figure 2.5: Trie index built over out motivating example of table 2.1

total ordering of terms within $\mathcal{V}_S$, i.e. $\mathcal{ORDER}(t_{13}) = 13$. Initially the set of terms are mapped to the their corresponding rank and sorted in decreasing rank order (t_1 is thought to be the highest ranked term of $|\mathcal{V}_S|$). Top down traversal starts from the root. The algorithm first considers term t_2 of subscription S_6. Term t_2 indicates that the path corresponding to term t_2 must be followed and such the algorithm proceeds. After processing the first term of subscription S_6, the second term t_3 from the sorted list is considered. However, no such path exists(recall that we have only indexed subscriptions S_1 *to* S_5) so a new node for term t_3 is created and inserted into the Trie as a child of term t_2. Finally, term t_{13} of S_6 is processed. Once again, since no such path exists term t_{13} is created and inserted as of t_2. Having consumed all of the terms of the subscription, S_6 is assigned to node labelled with t_{13} where the top down traversal of the Trie concluded. □

Construction time requirements

When adding a subscription, Algorithm, 5 initially sorts the terms of the subscription according their rank (set up phase). Next, it iterates through all of the terms and performs a top down traversal on the index (traversal phase). Hence, the time needed to insert a particular subscription is equal to:

$$Time(ADD\_SUBSCRIPTION) = Time(Set\_up) + Time(Traversal) \quad (2.10)$$

Since sorting can be performed in $O(N * \log N)$ [25] the time required for the set up phase is equal to:

$$Time(Set\_up) = |S_i|_{AVG} * \log |S_i|_{AVG} \quad (2.11)$$

When adding a subscription, for every term in the subscription, a lookup is performed to determine if the corresponding child node exists. If $Time(Look\_up)$ denotes the time required to perform such an operation, then the time required for traversal is equal to:

$$Time(Traversal) = |S_i|_{AVG} * Time(Look\_up) \quad (2.12)$$

If we suppose that $Time(Look\_up)$ can be performed in O(1), then adding a subscription into the Trie index can be accomplished in:

$$Time(ADD\_SUBSCRIPTION) = |S_i|_{AVG} * (1 + \log |S_i|_{AVG}) \quad (2.13)$$

Clearly, construction time is independent to the number of subscriptions currently indexed ($O(|S_i|_{AVG} * (1 + \log |S_i|_{AVG})$). Thus, adding subscriptions into the Trie index is scalable.

2.3.2 Trie Index - Matching

In order to match an news item I_k against the set of stored subscriptions, a top down traversal on the Trie index is performed. Initially, we obtain the sequence of the terms of I_k performing the same pre-processing the set of subscriptions where subject to. Traversal starts from the index root. Algorithm 6 illustrates the case. At each traversal step, the paths corresponding to all of the terms of I_k, whose ranks are superior to that of the term assigned to the currently considered node are followed (lines 4-9). Matching subscriptions are reported as the algorithm proceeds; for every node visited the subscriptions stored at that specific node are returned(lines 1-3).

Algorithm 6: $TRIE\_MATCH(TNode, [t_1...t_n])$

Require: $TNode$: the current trie node
Require: $[t_1...t_n]$: the sequence of terms

1: **if** $TNode$ contains subscriptions **then**
2: $S_{MATCHED} \leftarrow S_{MATCHED} \cup \{S_i | S_i \in TNode\}$
3: **end if**
4: **for all** term $t_j, j \in [1...n]$ **do**
5: $childNode \leftarrow$ get child for term t_j
6: **if** $childNode \neq NULL$ **then**
7: $TRIE\_MATCH(childNode, [t_{j+1}...t_n])$
8: **end if**
9: **end for**

Example 2.7

Suppose that we want to match news item $I_k = \{t_2, t_1, t_4\}$ over the index of Figure 2.5. Pre-processing I_k would result to the sequence $T_{SEQ} = [t_1, t_2, t_4]$. Top down traversal starts from the index root. Initially, term t_1 is considered. Since such a term exists in the index the algorithm proceeds to the specific node. When at the new node the sequence of terms is updated to $T_{SEQ}.1 = [t_2, t_4]$. While at term t_1 the algorithm considers rank t_2 of $T_{SEQ}.1$. Since such a path exists, it is followed and $T_{SEQ}.1$ is further updated to $T_{SEQ}.1.1 = [t_4]$. In a similar manner, after processing the final term t_4 of $T_{SEQ}.1.1$, the algorithm proceeds to term t_4 and returns subscription S_1. Back in the root, having processed term t_1 of T_{SEQ}, the algorithm considers the next term namely t_2. Since such a path from the root exists, the algorithm then proceeds to term t_2 of the Trie and updates T_{SEQ} to $T_{SEQ}.2 = [t_4]$. Once again the algorithm follows the specific path to term t_4. Subscription S_5 is reported as matched. Finally the algorithm considers the last term t_4 of T_{SEQ} and concludes since no such path exists to be followed. □

The count based approach needs to maintain a counter when matching a news item in order to ensure the conjunctive matching semantics applied. The Trie index structure on the other hand does not require such a mechanism. The reason being is that due to the inherent nature of how subscriptions are organized as a rooted tree, counting is performed implicitly as the matching algorithm progresses. Every time an edge is followed and a new term of the Trie

is further considered, the matching algorithm implicitly increments a 'single' counter for all subscriptions present in the same sub Trie rooted at that particular term. Consider for example the paradigm of figure 2.5 and suppose we want to match news item $I_k = \{t_1, t_2\}$. Further suppose that the algorithm has processed both of the terms of I_k. Since the length of the path from the root to the current term is equal to two, for both subscriptions S_3 and S_4 stored at descendant nodes of term t_2, the algorithm has implicitly counted two satisfied terms.

2.3.3 Trie Analytical Model

The memory requirements of the Trie index are directly related to the number of nodes (terms) it occupies. Towards that end, we present an upper bound on the total term occurrences for the Trie, based on the Pascal's triangle construct [20]. More precisely, we argue that, the occurrences per term of a complete Trie, can be modelled as a Pascal's Triangle. By exploiting several properties associated within, we then propose an analytical model for the memory requirements of the Trie with respect to a specific vocabulary and subscription size.

Trie as a Pascal Triangle

In mathematics, a Pascal's triangle is a triangular array of the binomial coefficients in a triangle. The triangle can be represented as an N by N matrix where:

- $T[0][j] = 1, \forall j \in [0, N-1]$
- $T[i][j] = \sum_{k=0}^{i-1} T[k][j-1]$

To better demonstrate the concept consider the example of Table 2.4 which depicts a Pascal's triangle as a 4 by 4 matrix. Observe that, all elements of the first column are equal to one. In addition:

$$T[1][1] = \sum_{k=0}^{0} T[k][0] = T[0][0] = 1$$

$$T[3][1] = \sum_{k=0}^{2} T[k][0] = T[0][0] + T[1][0] + T[2][0] = 1 + 1 + 1 = 3$$

Table 2.4: Pascals Triangle (4,4)

1	0	0	0
1	1	0	0
1	2	1	0
1	3	3	1

Following are several properties of the Pascal's Triangle that will be used in the sequel where we devise the analytical models. We will use the notation C_n^k to denote the number of k-combinations of n elements:

Property 2.3.1. *If we represent the Pascal's Triangle as a N by N matrix (let $\mathcal{T}$ denote the Triangle) then, for each element of the matrix it holds that: $\mathcal{T}[i][j] = C_i^j$*

Property 2.3.2. *A direct implication of Property 2.3.1 is that the sum of a column k in a triangle with N rows is equal to: $\sum_{n=0}^{N-1} C_n^k$*

Property 2.3.3. *The sum of row n in a Pascal's Triangle is equal to* 2^n

Lemma 2. *Consider a vocabulary of size* $|\mathcal{V}_S| = N$ *and that the terms of* $\mathcal{V}_S$ *are ranked from 0 to N-1. Let* $occ_l(t_j)$ *denote the number of occurrences of term* t_j *in level* l *of the Trie. If* $\mathcal{T}$ *denotes the Pascal's Triangle represented as an N by N matrix then:* $occ_l(t_j) = \mathcal{T}[j][l]$

Proof. By construction, a term with rank r will be a child for each term with rank m superior to its rank ($m < r$). Considering level l, the number of occurrences of term t_r is equal to the sum of occurrences of every term t_m in level $l-1$ for which it holds $m < r$. This is equal to C_r^l. However, according to Property 2.3.2, we know that $\mathcal{T}[r][l] = C_r^l$. Hence, $occ_l(r) = \mathcal{T}[r][l]$ □

According to lemma 2, we can represent the complete Trie with a vocabulary of size $|\mathcal{V}_S| = N$ as Pascal's Triangle. If we use the matrix representation for the Triangle then, rows will correspond to the terms and columns to the levels of the Trie. This is better demonstrated by example Figure 2.6 below. As we can see, every term of the vocabulary can exist only once at the first level (L_0). The first column of the Pascal matrix exposes this fact. Term t_2 appears in level L_1 2 times, that is once for every term of level L_0 with rank superior to 2 (t_0 and t_1). Observe that this is equal to the third row of the second column of the Pascal's matrix. Clearly, the number of occurrences of a particular term t_j of the Trie is equal to the sum of all occurrences of t_j for every level (sum of row j). In a similar manner, the number of terms assigned to level L_l is equal to the sum of all occurrences of every term for that specific level (sum of column l).

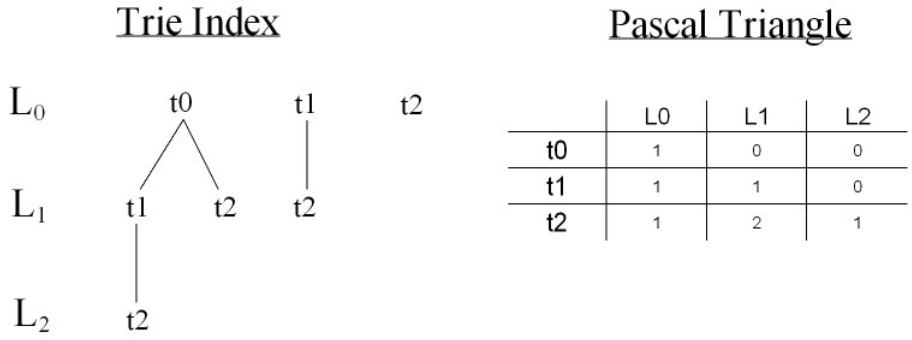


Figure 2.6: Complete Trie and corresponding Pascal Triangle representation

With the use of the Pascal matrix we can compute the total terms the complete Trie contains, by summing the occurrences for each term of every level:

$$Total\ Terms = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} \mathcal{T}[i][j] \tag{2.14}$$

Memory Requirements

Although equation 2.14 defines the total terms a complete Trie can occupy, it only takes into consideration the vocabulary $|\mathcal{V}_S|$ of the subscriptions. This implies a Trie with a maximum path of size $|\mathcal{V}_S|$. In the setting of web syndication where we expect small subscription size this is however, far from true. We suppose that $|S_i|_{AVG}| << |\mathcal{V}_S|$. Obviously, the length $|S_i|_{AVG}$ of a subscription S_i bounds the occurrences of a term to the first $|S_i|_{AVG}$ levels of the complete Trie. As table 2.5 illustrates, this 'trimmed' sub Trie, corresponds only to a part of the Pascal's triangle.

Table 2.5: Partial Pascal's Triangle

	l_0	l_1	l_2	-	$l_{\lvert S_i\rvert_{AVG}-1}$
t_0	1	0	0	-	0
t_1	1	1	0	-	0
t_2	1	2	1	-	0
-	1	-	-	-	-
t_{N-1}	1	N-1	$\sum_{k=0}^{N-2} k$	-	$\sum_{k=0}^{N-1} occ_{\lvert S_i\rvert_{AVG}-1}(t_k)$

Taking into account the cut-off effect of subscription size and Property 2.3.2, for a given vocabulary of size $|\mathcal{V}_S| = N$ the total terms occupied by the Trie are equal to:

$$TrieTerms = \sum_{k=0}^{|S_i|_{AVG}-1} \sum_{n=0}^{N-1} C_n^k \tag{2.15}$$

Upper Bound on term occurences for the Trie

Equation 2.15 bounds the total terms of the Trie with respect to a given subscription and vocabulary size. However, given the current setting this bound is practically unreachable. Towards that end, we will devise an refinement, based on the findings of the previous sub section on the Pascal's Triangle, that additionally considers term distribution information. Let us first define the number of expected terms for a given level.

Lemma 3. *Let $P(t_j \in \mathcal{S})$ denote the probability a term t_j exists within a subscirption of $\mathcal{S}$. If we use $P_l(t_j)$ to denote the probability we find $occ_l(t_j)$ (see Lemma 2) times term t_j in level l then:*

$$P_l(t_j) = \sum_{k=0}^{j-1} P_{l-1}(t_k) * P(t_j \in \mathcal{S}) * \left[\sum_{n=j+1}^{|\mathcal{V}_S|-1} P(t_n \in \mathcal{S})\right]^{|S_i|_{AVG}-l-1}$$

, where:

$$P_0(t_j) = P(t_j \in \mathcal{S}) * \left[\sum_{k=j+1}^{|\mathcal{V}_S|-1} P(t_k \in \mathcal{S})\right]^{|S_i|_{AVG}-1}$$

Proof. We will prove Lemma 3 by induction. Let us first consider the first level (l_0) of the Trie. For the first level we know that a term exists iff it is a prefix. For a term t_j to be a prefix for a subscription of length $|S_i|_{AVG}$, t_j needs to exist in the subscription, and all other $|S_i|_{AVG} - 1$ terms need to have ranks inferior to j. We know that the probability to obtain a term inferior to t_j is equal to:

$$P(obtain\ a\ term\ inferior\ to\ t_j) \quad = \quad \sum_{k=j+1}^{|\mathcal{V}_S|-1} P(t_k \in \mathcal{S}) \tag{2.16}$$

Hence, the probability for term t_j to be a prefix of S_i is equal to:

$$\begin{aligned} P(t_j\ prefix\ of\ S_i) &= P(t_j \in \mathcal{S}) * \prod_{k=1}^{|S_i|_{AVG}-1} P(obtain\ a\ term\ inferior\ to\ t_j) \\ &= P(t_j \in \mathcal{S}) * \left[\sum_{k=j+1}^{|\mathcal{V}_S|-1} P(t_k \in \mathcal{S})\right]^{|S_i|_{AVG}-1} \end{aligned} \tag{2.17}$$

So, $P_0(t_j)$ can be computed as:

$$\begin{aligned} P_0(t_j) &= P(t_j\ prefix\ of\ S_i) \\ &= P(t_j \in \mathcal{S}) * \left[\sum_{k=j+1}^{|\mathcal{V}_S|-1} P(t_k \in \mathcal{S})\right]^{|S_i|_{AVG}-1} \end{aligned} \tag{2.18}$$

We will next compute the expected number of terms for the second level using level l_0. To better demonstrate the approach, observe Table 2.5 and lets consider the number of times term t_2 appears in level l_1. We anticipate t_2 to exist two times in level l_1 if, t_0 and t_1 of l_0 exist and t_2 is a prefix(t_2 needs to be a prefix after we discard one of the terms, since we now consider level l_1). Hence, for a term t_j we can say that:

$$\begin{aligned} P_1(t_j) &= \sum_{k=0}^{j-1} P_0(t_k) * P(t_j\ prefix\ of\ sub\ with\ size\ |S_i|-2) \\ &= \sum_{k=0}^{j-1} P_0(t_k) * P(t_j \in \mathcal{S}) * \left[\sum_{n=j+1}^{|\mathcal{V}_S|-1} P(t_n \in \mathcal{S})\right]^{|S_i|_{AVG}-2} \end{aligned} \tag{2.19}$$

Suppose that we have computed the probability $P_{l-1}(t_k)$ for every term t_k of level $l-1$ (induction step). In order for a term t_j to appear $occ_l(t_k)$ times in level l then it has to hold that, every term t_m $(m < j)$ of level $l-1$ exists (that is $P_{l-1}(t_m)$) and that t_j is a prefix of a subscription of size $|S_i|_{AVG} - (l+1)$. Since we have computed $P_{l-1}(t_m)$ for all $t_m \in \mathcal{V}_S$ we can say that:

$$\begin{aligned} P_l(t_j) &= \sum_{k=0}^{j-1} P_{l-1}(t_k) * P(t_j\ prefix\ of\ sub\ with\ size\ |S_i|-(l+1)) \\ &= \sum_{k=0}^{j-1} P_{l-1}(t_k) * P(t_j \in \mathcal{S}) * \left[\sum_{n=j+1}^{|\mathcal{V}_S|-1} P(t_n \in \mathcal{S})\right]^{|S_i|_{AVG}-l-1)} \end{aligned} \tag{2.20}$$

□

Matching Requirements

The time required to match an news item I_k against the set of stored subscriptions is proportional to the number of terms visited. In a worst case scenario, the algorithm would have to follow all the paths corresponding to every subset of I_k ($\mathcal{O}(2^{|I_k|})$). In such a case, the sub Trie corresponding to the traversal performed upon matching, is equal to the Trie built over the terms of I_k. However, in order to obtain such a complexity the algorithm has to explore the

Trie data structure up to level $|I_k|_{AVG}$ which implies that $|I_k| < |S_i|$. On the contrary, we expect $|I_k|_{AVG} > |S_i|$; subscription size bounds the worst case. If we represent news item I_k as a Pascal's Triangle (observe Table 2.6) in the worst case the number of terms required by the matching algorithm to be visited will be equal to:

$$VisitedTerms = \sum_{k=0}^{|S_i|_{AVG}|-1} \sum_{n=0}^{|I_k|_{AVG}-1} C_n^k \tag{2.21}$$

Table 2.6: Pacal's Triangle over new item I_k

	l_0	l_1	l_2	-	$l_{\|S_i\|_{AVG}-1}$
t_0	1	0	0	-	0
t_1	1	1	0	-	0
t_2	1	2	1	-	0
-	1	-	-	-	-
$t_{\|I_k\|_{AVG}-1}$	1	$\|I_k\|_{AVG}-1$	$\sum_{k=0}^{\|I_k\|_{AVG}-2} k$	-	$\sum_{k=0}^{\|I_k\|_{AVG}-1} occ_{\|I_k\|_{AVG}-2}(t_k)$

2.3.4 Trie index remarks

As opposed to the flat vocabulary considered by the Count index, the covering relations captured by the Trie index allow frequent terms appearing as common prefixes to be organized in a fine-grained hierarchical search space which in turn results to faster mathching. Moreover, the tree structure alleviates the need for maintaining an additional structure as counting is now performed implicitly. As Table 2.7 illustrates, w.r.t. the memory requirements, the size of the subscriptions serve as a cut-off on the depth of the Trie. As for matching, the total number of paths followed when matching is bound to the size of the news item.

Table 2.7: The Trie as Pascal's Triangle: summary

	l_0	l_1	l_2	-	$l_{\|S_i\|_{AVG}-1}$	-	N-1
t_0	1	0	0	-	0		
	-	0					
t_1	1	1	0	-	0		
	-	0					
$-$	1	-	-	-	-		
	-	0					
$t_{\|I_k\|_{AVG}-1}$	1	$\|I_k\|_{AVG}-1$	$\sum_{k=0}^{\|I_k\|_{AVG}-2} occ_1(t_k)$	-	$\sum_{k=0}^{\|I_k\|_{AVG}-2} occ_{\|S_i\|_{AVG}-2}(t_k)$	-	0
-	1	-	-	-	-	-	0
t_{N-1}	1	$N-1$	$\sum_{k=0}^{\|I_k\|_{AVG}-2} k$	-	$\sum_{k=0}^{N-1} occ_{\|S_i\|_{AVG}-2}(t_k)$	-	1

Worst case (memory and matching):

$$TrieTerms = \sum_{k=0}^{|S_i|_{AVG}|-1} \sum_{n=0}^{N-1} C_n^k$$

$$VisitedTerms = \sum_{k=0}^{|S_i|_{AVG}|-1} \sum_{n=0}^{|I_k|_{AVG}} C_n^k$$

Prefix Sharing

It is expected that users with similar interests to issue similar subscriptions. It is thus likely for a great number of subscriptions to share common terms. Using the Trie index structure proposed, in contradistinction to list structures employed by the inverted file case of the Count based index could result to more compact representations with regard to the total number of nodes each case occupies (Trie nodes versus inverter file subscription postings nodes). However it is not always the case that two subscriptions sharing a common subset of terms to also share a path within the Trie index.

Lemma 4. *Any two subscriptions S_i and S_j share a common path if and only if* $\forall t_k \in \{S_i \cup S_j\} - \{S_i \cap S_j\}, \forall t_l \in \{S_i \cap S_j\} \rightarrow \mathcal{ORDER}(t_l) < \mathcal{ORDER}(t_k)$.

Recall, that the Trie is an ordered data structure. Hence, two subscriptions share a common path if and only if ordering the terms according to rank order results to a common prefix. The number of nodes they share is equal to the size of their common prefix. To better demonstrate how this applies consider the example 2.8 below.

Example 2.8

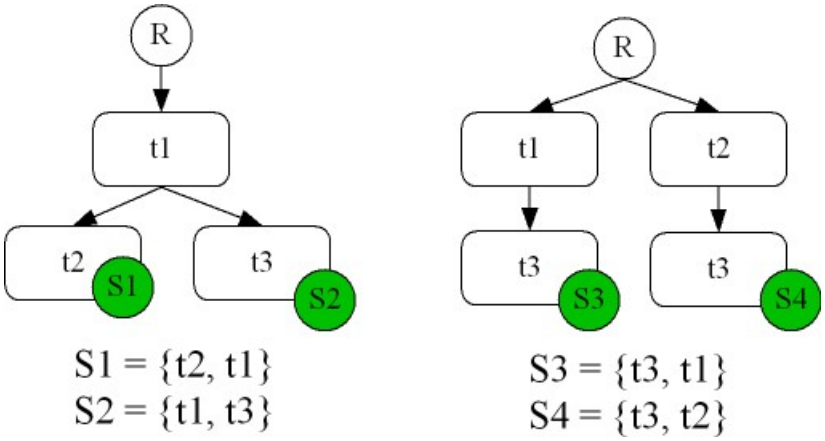


Figure 2.7: Example of prefix sharing

Consider for example figure 2.7. In the first case on the left subscriptions $S_1 = \{t_2, t_1\}$ and $S_2 = \{t_1, t_3\}$ are inserted into the Trie index. Both of the subscriptions S_1 and S_2 share the common subset $S_{COMMON} = \{t_1\}$. Obviously it holds that $\mathcal{ORDER}(t_1) < \mathcal{ORDER}(t_2)$ and $\mathcal{ORDER}(t_1) < \mathcal{ORDER}(t_3)$; their common subset is also a common prefix. This property is also reflected in the index as term t_1 exists only once. In the second example on the right hand side of the

of figure 2.7 however where subscriptions $S_3 = \{t_3, t_1\}$ and $S_4 = \{t_3, t_2\}$ are indexed this is not the case. Despite the fact that both of them share the common term t_3 since it is not a common prefix no node sharing will be activated ($\mathcal{ORDER}(t_3) > \mathcal{ORDER}(t_2)$ (and $\mathcal{ORDER}(t_3) > \mathcal{ORDER}(t_1)$)). □

Trie Characteristics

As Figure 2.8 illustrates, the morphology of the Trie index is affected by subscription size, vocabulary size, and the relation between the actual ordering of the subscription terms and the ordering used for construction.

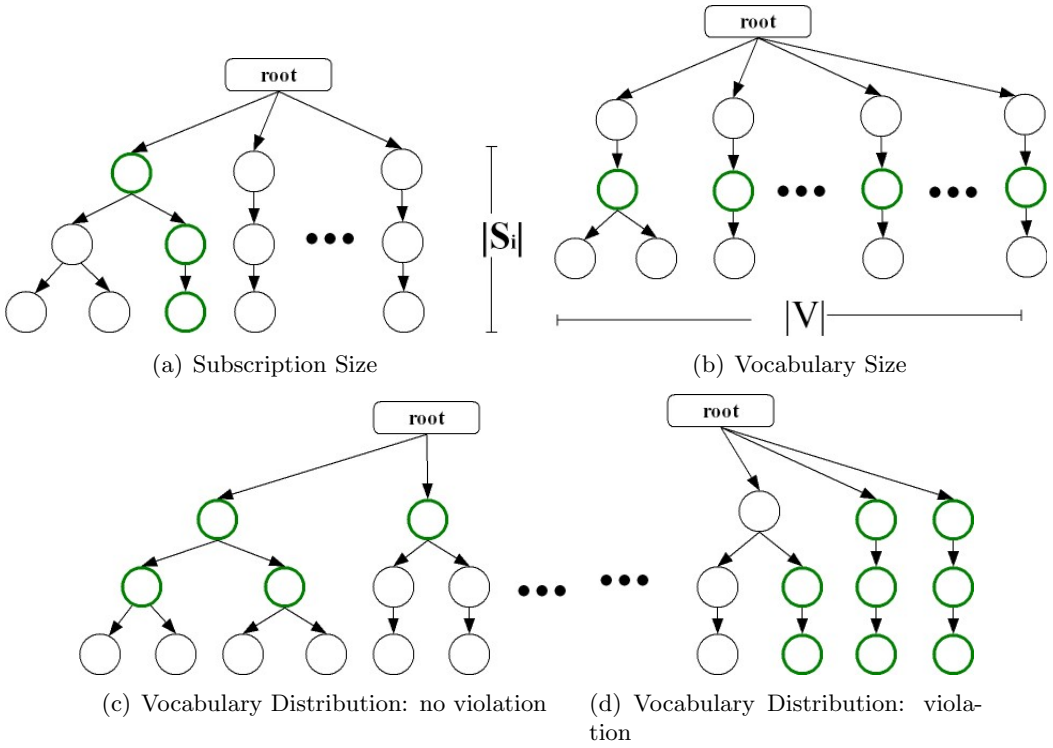
Subscription Size: Impacts the height of the Trie. Recall that when inserting a subscription, Algorithm 5 performs a top down traversal and in a worst case would insert $|S_i|$ terms in the Trie. Hence, greater subscription sizes will produce *deeper* Tries.

Vocabulary Size: Impacts the width of the Trie. Greater vocabulary sizes will produce *broader* Tries.

Term Ordering: If actual frequency based term ordering in subscriptions does not violate the initial ordering considered for construction, then highly ranked terms will appear as intermediate nodes (with several children), as a *left-structured* Trie. These highly ranked terms will not appear in great numbers since due to construction constraints highly ranked terms can appear only a few times in the Trie (i.e. the highest ranked term could only appear once, the second highest twice, etc.)and in addition prefix sharing will take place, as these highly ranked terms will also form common prefixes. In the opposite case, a great number of lowly ranked terms (low ranked term are allowed to appear more times than highly ranked terms) will appear as intermediate nodes (with a unique child) of a *right-structured* Trie. As opposed to the former case, prefix sharing will not be activated.

In the worst case (no shared prefixes amongst the set of indexed subscriptions) the total terms occupied by the Trie will be equal to the number of terms in the subscriptions. Apparently, the Trie results to more compact representations when subscriptions share common prefixes. Hence, memory wise it is of best practice to suppose an ordering for construction that does not violate the actual frequency based ordering of the terms in the subscriptions. As opposed to multiple term occurrences of the Trie, which are minimized when the actual frequency based ordering of the subscription terms does not violate the initial ordering used for construction, best performance with respect to matching is achieved if we consider reverse ordering (i.e. the most frequent term to be low ranked). Recall that the time required when matching depends on (a) the number of nodes visited and (b) the amount of reduction in the item suffix which will be further checked in each step. Hence if reverse ordering is applied then (a) more subscriptions will be stored at sub-Tries of lower ranked terms which in turn are expected to be looked up less and (b) the expected reduction in the size of the suffix will be larger (see Figure 2.9).

Apparently, there exists a trade-off between the gain in memory and matching time with respect to the relation between the ordering considered for construction and the actual frequecy-based ordering of the subscription terms.



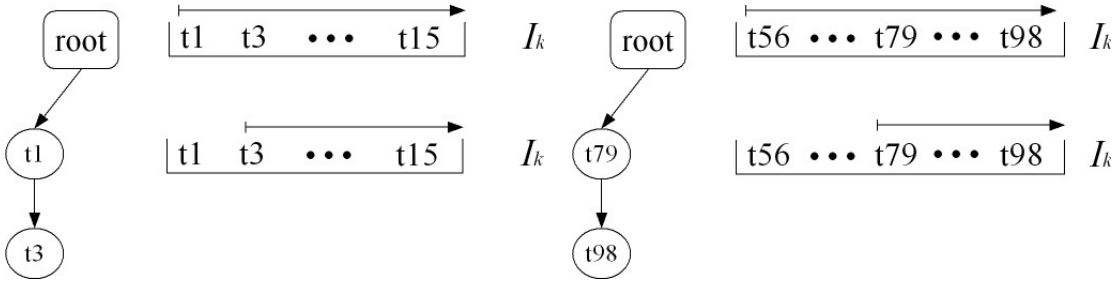
(a) Subscription Size

(b) Vocabulary Size

(c) Vocabulary Distribution: no violation

(d) Vocabulary Distribution: violation

Figure 2.8: Trie Morpholgy



(a) Actual ordering: no violation

(b) Actual ordering: violation

Figure 2.9: Trie Matching: suffix reduction

Chapter 3

Implementing the Subscription Indexes

In order to perform experimental evaluation and verify the analytical findings presented in previous chapters, a prototype term based publish subscribe system was developed. The integral component of our system is its filtering engine. The core component of our filtering engine is a subscription index. The input to the system consists of subscriptions which are indexed and items that are matched. The system outputs the set of matched subscriptions matched for a given news item.

An overview of the systems interface is illustrated in Figure 3.1. Adding or deleting a particular subscription is performed via the addSubscriber() and deleteSubscriber() interface. Matching is performed with the use of the matchItem() interface. Within our system, subscription and item terms are encoded as integers which reflect their frequency rank and serve as identifiers. Every subscription (news item) is assigned to a unique identifier and a list of integer term ids. Subscription identifiers are provided by the system. The set of matched items for a specific subscription can be obtained via the getMatchedItems() output interface. Current implementation incorporate both the Count based and the Trie indexes. Implementations were coded and tested within the Java platform.

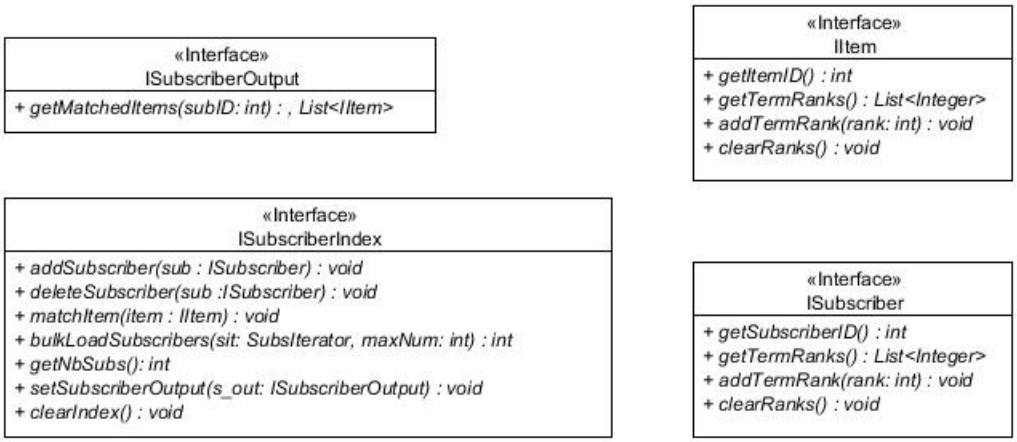


Figure 3.1: Prototype Term Based Pub/Sub interface

3.1 Count based index

Two implementations of the Count based index have been considered [19]. The first one refered to as *Simple* implements all structural components via standard Java constructs. The second implementation, refered to as *Compact*, considers custom implementations that extend the standard Java structures for both the Inverted file and Counter to enable more compact representations.

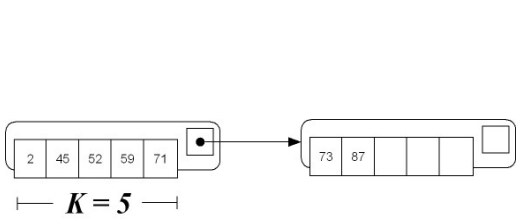
3.1.1 Simple Count based index

In the *Simple* count based index, the inverted file is implemented as a standard Java Hash Table. Every entry of the hash table contains the term id and a reference to the list of corresponding subscription postings. Subscription postings are stored in a standard Linked List of integer wrapper objects. The counter is implemented as an array which doubles it's capacity whenever a given threshold of occupacy is exceeded.

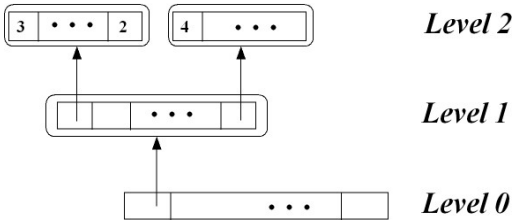
3.1.2 Compact Count based index

The *Compact Count* based index is illustrated in Figure 3.2(c). This implementation maps every term of the vocabulary to its inverted set of subscription postings via a hash table of fixed capacity C. The standard hash table implementation of java is extended for this purpose. Hash table collisions are resolved with the use of custom linked lists. Each node of the collision list contains an integer term identifier, a pointer to the list of subscriptions corresponding to that particular term and finally a pointer to the next node of the collision list. Subscription postings corresponding to a particular term, are customly implemented as array lists. Each node of the list contains an integer array of constant size K and a pointer to the next posting list node. Figure 3.2(a) illustrates the case. The reason why such an approach was preferred over the standard list implementation of java's JDK is that it results to more compact representations. Java standard list implementations store objects. Hence, if such a solution was devised then for every subscription id stored within the posting lists an Integer wrapper object would need to be created. Since java objects require more memory than that of the integer primitive type, storing subscription postings using the custom approach results to smaller overall memory requirements for the inverted file.

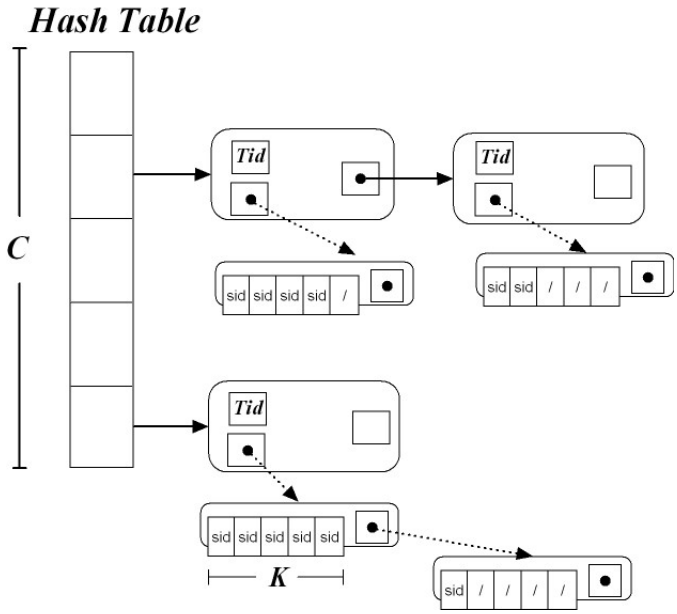
Whenever a new subscription is added to the inverted file index, a new entry within the count structure with a value equal to the size of the subscription is inserted. Hence, the data structure employed for counting has to be dynamic. That is to say, that it should grow along with subscription insertion. A second requirement for the count structure is to provide fast access to the number of terms a particular subscription contains given the subscription's id. Efficient matching requires obtaining the count value for a particular subscription to be done in a timely manner. One way to trivially implement such a structure is to suppose an integer array of specific initial capacity and accommodate dynamicity by doubling the capacity on demand according to pre-defined policy (i.e. when the size of the array has reached its capacity). The size of a particular subscription could be obtained by directly indexing the array via the subscription identifier. However, the basic disadvantage of this approach is that it does not handle well subscription deletions. Consider for example that the index has been loaded with 1,000,000 subscriptions and further suppose that 500,000 of them where deleted after insertion was performed. Given that we directly index the count data structure via subscription identifier, the size of the count structure should be at least equal to the largest subscription id currently indexed. This leads to potential memory waste.



(a) subscription postings list implementation



(b) two level counter for storing the total terms of a subscription



(c) inverted file implementation

Figure 3.2: Compact Count index implementation

To confiscate the above observation within current development the count data structure has been implemented as a *two level* index. The main idea is to employ a data structure that partitions subscriptions into groups according to their id. Figure 3.2(b) depicts the case. The entry point for that specific structure is the ground table. Every bucket of the ground level corresponds to a group of subscriptions. Group size depends on the partitioning applied. Each bucket of the ground level contains a pointer to the second level where subscriptions are further segmented into more fine grained groups. All subscription sizes are stored within the last level of the data structure. To demonstrate how indexing is applied suppose we want to obtain the number of terms subscription with id 59,123 contains. Further assume that the size of the ground level is $|Level_0| = 10$, the second level is of size $|Level_1| = 20$ and finally the third level has a size of $|Level_2| = 50,000$. Since $59,123/(|Level_1| * |Level_2|) = 0$ we know that the particular subscription is contained within the first group of subscriptions and proceed to the array the first bucket of the ground table points to. Next, we consider the index $59,123\%(|Level_0|*|Level_1|*|Level_2|) = 59,123$. In a similar manner since $59,123/(|Level_2|) = 1$ we proceed to the array of level two that the pointer of second bucket of the first level points to. Finally we index the second level with $59,123\%(|Level_2|)$ and return the specific value.

The inverted file structural behaviour could be controlled via the parameters C and K which correspond to the hash table capacity and the size of the custom array list implementation used

for storing subscription postings respectively. Parameter C affects vocabulary storage. Larger capacity results to more entries for the hash table. Assuming a good hash function (i.e. a hash function that evenly distributes elements within all buckets) this would result however to smaller collision lists. Parameter K on the other hand impacts the storage of subscriptions. Larger values of K result to smaller numbers of subscription postings list nodes. This leads to more compact representations since less postings list node objects would need to be created. Nevertheless, this can result to memory waste since it might be the case that many empty entries would exist. For example if we consider a size of $k = 100$ and assume only additions, then for every term within the vocabulary, the last posting list node assigned to that specific term would on average contain 50 unoccupied integer slots. The best performance for both parameters was achieved when C and K where set to 50% and 25 respectively (see Tables 3.1 and 3.2) [19]. Experimental evaluations in next chapter consider these specific values.

The best combination of parameter values, as a simple experimental evaluation on the Count based index was performed by varying both C and K. Tables 3.1 and 3.2 summarize the results.

Table 3.1: influence of parameter C(capacity)

Capacity	**Memory(MB)**	**Index Insertion**	**Index matching**
50%	107.01	0.0427	5.3988
60%	109.93	0.0428	5.4241
70%	105.18	0.0424	5.3872
80%	105.24	0.0429	5.3891
90%	109.60	0.0417	5.3969
100%	109.74	0.0426	5.5109

Table 3.2: memory in MB for different values of K

	K=10	**K=25**	**K=50**	**K=100**
500,000	53.01	51.10	62.61	83.1
1,000,000	92.72	88.83	104.68	132.23
4,000,000	324.29	275.51	301.41	343.95
9,000,000	681.71	492.71	538.61	633.29

3.2 Trie index

Many different approaches have been devised for implementing the Trie data structure [34, 5]. More specifically, a Trie could be implemented as a linked list, array or tree. In this work we consider a hash tree based [35] implementation for the index.

3.2.1 Simple Trie

The integral components of a Trie node as described in earlier chapters are, the term rank that identifies a particular node, the structure that stores the set of subscriptions assigned to the node, and finally the child structure used for storing the set of children the node under consideration contains. A first implementation of the Trie index referred to hereafter as *Simple Trie*, distincts between internal and leaf nodes. An internal node contains: a Java primitive *int* type for storing the term rank assigned to the node, a standard Java *Linked List* for storing the subscriptions of the node, and finally a Java *Hash Map* for storing the childs. For internal nodes, the initial capacity of the Hash Map is set to 2. As opposed to internal nodes, leaf nodes

only contain an int type for the rank and a standard Linked List for the subscriptions. Figure 3.3 depicts the index built over the example subscription set of Table 2.1. Internal nodes are labeled with *(I)*. Coloured nodes correspond to subscription list nodes.

The reason that motivated this approach (distinction between internal and leaf nodes) lies in the fact that, since we expect a rather large vocabulary, we anticipate subscriptions to be distributed to a large number of different Trie nodes. Hence, it is expected for the Trie to contain many leafs.

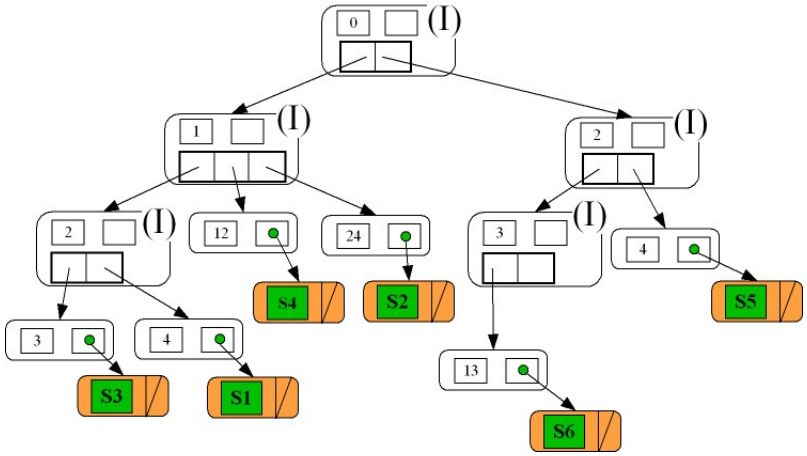


Figure 3.3: Simple Trie implementation; internal nodes are labelled with (I); subscription lists nodes are coloured.

3.2.2 Compact Trie

A second implementation that extends the *Simple Trie* and further exploits the anticipated impact of the workload on the Trie structure, with respect to the number of subscriptions and childs per node, was also considered. The *Compact Trie* as presented in the following, considers a variety of different type node as well as path compression.

Depending on how a particular node n_k stores subscriptions, it can be categorized into one of the following three classes: n_k is considered as a class 'a' node if no subscriptions are assigned to it; if only one subscription is stored at n_k then it is considered to be a class 'b' node; finally if more than one subscriptions are assigned to node n_k then that particular node is regarded as a class 'c' node. Within current implementation, class 'a' nodes do not contain a field for storing subscriptions, class 'b' nodes use the simple integer primitive type of the JDK to store the single subscription identifier assigned to n_k, and class 'c' nodes use the custom array implementation presented earlier with parameter K set to $K = 2$.

The nodes of the *Compact Trie* are further partitioned into groups based on the number of children they contain. As with the Simple Trie case, depending on weather a node n_k contains a child or not it can be classified as either internal or leaf node. The former are referred to as class 'II' nodes and the later as class 'I' nodes. If a node n_k is a class 'I' node then no field exists for storing child reference. On the contrary, if n_k is an internal node then, it contains an additional data structure for storing references to child nodes. Within current implementation, type 'II' nodes contain a standard Java hash map of initial capacity of two.

Based on the previous classification, the *Compact Trie* implementation considers five different types of nodes all of which are depict in table 3.3. Clearly, a *TypaIa* node could not exist, since if

a particular node is a leaf then by construction a subscription would be assigned to that specific node.

Table 3.3: Classification of nodes for Compact_Trie

Child Structure	**Subscriptions Structure**	**Node Type**
none	none	not applicable
	integer type	TypeIb
	custom array list	TypeIc
	none	TypeIIa
JDK hash map	integer type	TypeIIb
	custom array list	TypeIIc

As table 3.3 illustrates, for nodes containing more than one nodes the standard JDK hash map is employed for storing child references. However, such an approach could be considered wastefull, as we anticipate a great number of nodes to contain a single child, given the workload we consider. Hence, a better implementation concerning how children are stored should be devised. One solution that would result to a more compact representation would be to additionally consider a third classification that considers nodes with only a single child. Implementing such a node type would result to a node with only one field instead of a Hash Map for the single child. Doing so would result to single paths being trivially implemented as single linked lists. Although this would lead to more memory efficient Trie indexes since single references replace the hash map structure, a better solution would be to employ path compression.

Current development, considers path compaction for the Trie index. Single paths corresponding to multiple nodes are compacted into one single compact node. Each compact node is labelled with the set of terms corresponding to the labels the nodes that the non compacting scheme would contain. For every *primary* node type illustrated in table 3.3 a *compact* dual exists (i.e. TypeIIb(compact)). As opposed to primary node types (no compaction is considered) where term ranks are stored as integer types, compact nodes store the list of term ranks as java int arrays.

3.3 Synthetic data generation

In order to evaluate the indexes, both subscriptions and news items where syntetically generated. Algorithm 7 below ilustrates the procedure. The workload parameters that define the generated set and serve as an input in our synthetic data generation module are: a vocabulary of terms V with information about the frequency of each particular term, the total number N of elements generated, a generator *LGen* for determining the length of each element, and finally the distribution that the terms of the generated subscription(news item) corpus would obey.

As Algorithm 7 illustrates, when generating an element I, l distinct terms, with respect to the distribution *dist* under consideration, are independently drawn from the vocabulary V. The length specified by *LGen* could either be fixed for all subscriptions or follow a user specified ditribution. Three different sampling modes, referred to as *empirical*, *uniform*, and *anti-correlated* are applicable. If an empirical term distribution mode is considered then the terms are sampled from the vocabulary V according to the frequency distribution that the terms of the vocabulary follow (recall that V contains term-frequency mappings). If a uniform term distribution mode is selected to be applied, then the frequency distribution available for the terms is simply discarded, and sampling is done in a uniform manner over the whole vocabulary.

Algorithm 7: $Generator(V, N, LGen, Dist)$

Require: V : A vocabulary V containing term-frequency mappings.
Require: N : The total amount of subscriptions(news items) to be generated.
Require: $LGen$: Element length generator; outputs the length for each element
Require: $Dist$: Controls sampling mode; values in $\{empirical, anti-correlated, uniform\}$.

```
1: Γ ← {}
2: for i to N do
3:     I ← {}
4:     j ← 0
5:     l ← generate length acocrding to LGen
6:     while j < l do
7:         term ← select terms from V according to Dist
8:         if term ∉ I then
9:             I ← I ∪ {term};
10:            j ← j + 1;
11:        end if
12:    end while
13:    Γ ← Γ ∪ {I}
14: end for
15: return Γ
```

Finally, in the anti-correlated case, the elements are generated by sampling terms according to the inverse term frequency distribution provided with V. The procedure outputs the set of elements Γ[1] generated according to the input workload parameters provided.

We have to stress out here that, the vocabulary applied for synthetic subscription (news item) generation and the actual vocabulary of the generated subscriptions might be different. In order to differientate between the two, the notation $\mathcal{V}_S^U$ will be used hereafter to denote the 'universal' vocabulary used for generating the set of subscriptions. Apparently, depending on the parameters used by the generating module $\mathcal{V}_S^U \subseteq \mathcal{V}_S$.

[1]Stricktly speaking *Gamma* is a multi set as generation does not guarentee uniqueness of generated elements

Chapter 4

Experimental Evaluation

In this chapter we experimentally evaluate the performance of both the Trie and Count based indices studied in Chapter 2. In particular we are interested on understanding how the *memory* and *time* required to *build* the subscription index as well as the *time* required to match an individual news item against the set of stored subscriptions is affected by a number of crucial parameters in a web syndication setting. More precisely we ran a total set of 6 experiments, each of which targeted an individual parameter (see Table 4.1) of the web syndication workload we generated synthetically according to the method described in the previous Chapter. The two main questions of our investigation was (a) how the *morphology* of the two indexes is affected by different workload parameters, i.e. the terms frequency distribution, the size of the vocabulary, and the size of the subscriptions and (b) how Trie and Count morphology impacts the scalability and performance of the two indices for realistic characteristics of subscriptions and news items. All experiments were conducted on a quadcore CPU machine at 2.40GHz with 3GB of memory running Ubuntu 10.04.

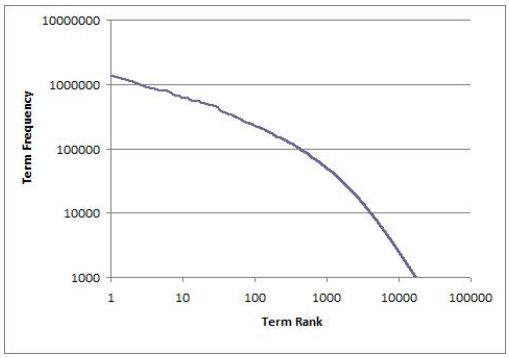
Table 4.1: Workload parameter values

Name	Description	Range
$\|\mathcal{S}\|$	Total number of subscriptions	$\{500000, 10000000\}$
$\|S_i\|$	The size of the subscriptions	$\{3, 6, 9, 12\}$ and empirical distr.
$\|I_k\|$	The size of the items	$\{5, 10, 20, 50\}$ and empirical distr.
$\|\mathcal{V}_S\|$	The size of the vocabulary	$\{10000, 100000, 800000\}$
$Sub\ Term\ Dist$	The distribution of the terms of the subscriptions	$\{empirical, anti-correlated, uniform\}$
$Item\ Term\ Dist$	The distribution of the terms of the items	$\{empirical, anti-correlated, uniform\}$

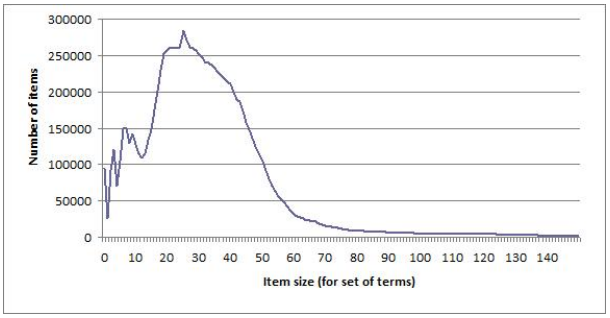
We assume that the size of subscriptions is close to the size of web search engine queries (see Section 1.1.3). Hence, the empirical subscription size distribution used in the experiments throughout this Chapter will refer to the size distribution of Figure 4.1(c) which was extracted from a series of search engine logs of three major search engine namely AltaVista, Excite and AlltheWeb [32].

The empirical vocabulary distribution used for generating the set of subscriptions (and news items), which corresponds to 10,799,285 news items acquired from 8,125 RSS feed [21] is depict in Figure 4.1(a). The actual size of the vocabulary was 1,462,599, however as Table 4.1 illustrates our domain of interest for $|\mathcal{V}_S|$ was $\{10000, 100000, 800000\}$. Downsampling, was performed by randomly picking in a uniform manner the the number of terms that were of specific interest; i.e. if to obtain a smaller vocabulary of size $|\mathcal{V}_S| = 10,000$, from the initial vocabulary, 10,000 terms (and their corresponding frequencies) were randomly selected.

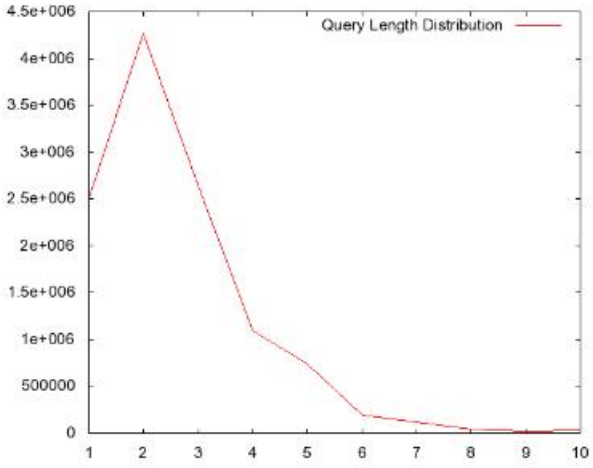
Finally, the empirical news item size distribution used for specifying the size of the synthetically generated news items, is depict in Figure 4.1(b) and corresponds to the length distribution the same set of 10,799,285 news items follow.



(a) Empirical vocabulary distribution



(b) Empirical news item size distribution



(c) Empirical subscription size distribution

Figure 4.1: Empirical distributions for vocabulary and subscription/item size

4.1 Impact of the Vocabulary Distribution

In this first set of experiments we wish to capture the impact of the term distribution on the structural characteristics of the two indexes. Measuring the number of nodes of the posting lists in the Inverted File (IF) as well as of the Trie provides us a better understanding as to how critical workload parameters such as terms' frequency distribution actually impact the memory requirements of the two indexes, and will allows us latter on to cross-validate our experimental findings. Two series of evaluations where conducted by considering three possible term frequency distributions, namely, empirical, anti-correlated, and uniform. The first one considers a smaller scale experiment on a simple implementation (see Sub Sections 3.1.1 and 3.2.1) of the indexes in order to better comprehend the behavior of the index structures under no optimization. The second series regards a full scale experimental evaluation on both the *compact* Count based and Trie index as presented in Sub Sections 3.1.2 and 3.2.2.

4.1.1 Evaluation on Simple Implementations

In this SubSection we are interested in studying the impact of the vocabulare distribution to the size of the postings list in the Inverted File. Furthermore, for the Trie index we are interested in understanding (a) how the Trie nodes increase as new subscriptions are added to the index, w.r.t. a given vocabulary distribution (b) how many Trie nodes are created w.r.t. to terms' rank and (c) what is the morphology of the resulting Trie, especially its out-degree distribution. The workload parameters where fixed to:

- $|\mathcal{V}_S| = 10,000$
- $|S_i|_{AVG} = 4$
- $|\mathcal{S}| = 500,000$

Inverted File - Posting List Size

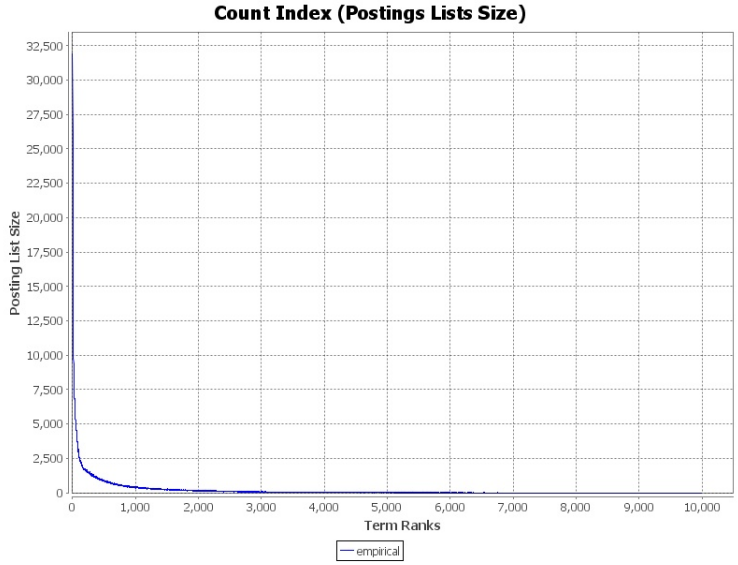
Figures 4.2(a) and 4.2(b) illustrate the results obtained with regard to the postings list size of the Inverted File (IF), when the uniform and empirical vocabulary distributions are considered. As expected, for both cases the underlying subscription term distribution is reflected within the inverted file structure. When the empirical distribution is considered only a few terms are assigned to large postings lists; many terms on the other hand are assigned to significantly smaller postings lists. When terms are uniformly distributed, the size of the lists is roughly equal to 200. This is reasonable, considering the specific workload parameters applied. Recall that we assume a vocabulary size of 10,000, a total of 500,000 subscriptions and the length of the subscriptions is equal to 4.

Figure 4.2(a) verifies our claims presented in Chapter 2, that the count based approach does not enable to prune efficiently the subscription search space when matching incoming news items. If both subscriptions and items follow the same skewed distribution (which is a realistic assumption), then a large number of subscription postings will need to be traversed in order to discover the matching subscriptions.

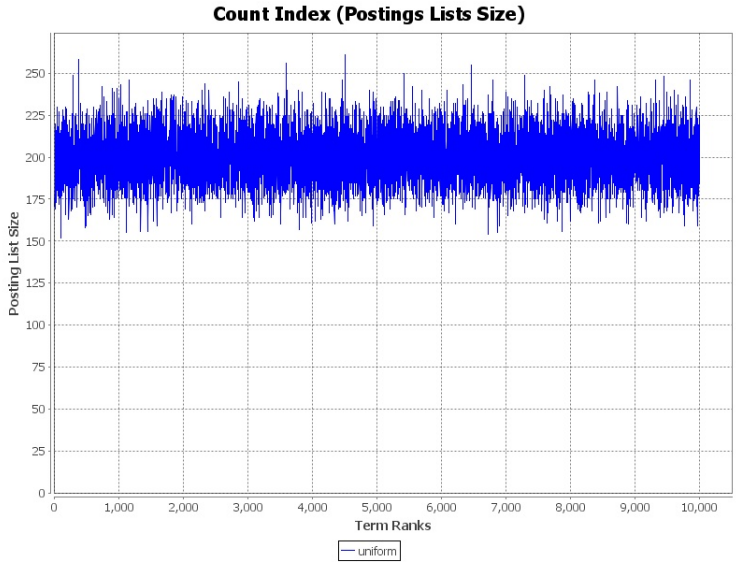
Trie - Increase rate of nodes

In Figure 4.3 we can see how the increase in the number of subscriptions indexed affects the total number of nodes constructed by the Trie, per different vocabulary distribution. The blue, black and red line of the plot concern a empirical, an uniform and an anti-correlated subscription term distribution respectively. As the results exhibit, the Trie index occupies the greatest amount of nodes ($\approx$ 1,500,000) when the uniform vocabulary distribution case is considered. The empirical distribution performs best node wise, as it only requires approximately 1,100,000 nodes (36% less than when compared to the uniform case). Finally, when subscription vocabulary distribution is anti-correlated, the total number of nodes occupied by the Trie index is approximately 1,450,000, roughly 3% less than the uniform case.

This behavior can be explained as follows. When an empirical distribution is considered, it is highly probable that a great fraction of the subscriptions will have common terms among the most frequent ones, hence, many subscriptions are expected to share common prefixes. This however, does not apply when an anti-correlated distribution is considered. Despite the fact that the anti-correlated case is also skewed, the probability that two subscriptions share the same prefix is low. This is due to the fact that skewness concerns mainly lower ranked terms.



(a) Empirical subscription vocabualry distribution



(b) Uniform subscription vocabulary distribution

Figure 4.2: Vocabulary distribution impact on Invered File (IF) of the Simple Count index: $|\mathcal{S}| = 500,000$, $|\mathcal{V}_S| = 10,000$, $|S_i|_{AVG} = 4$

The total number of posting list nodes that the Count based index would require, regarding the same workload, would be: $2,000,000$ ($|S_i|_{AVG} * |\mathcal{S}|$) nodes. In the worst case (uniform distribution) the Trie occupies $1,500,000$ nodes. If we consider the memory requirements in terms of the total number of nodes both indices require, a gain for the Trie would exist if and only if:

$$\frac{|trieNode|}{|postingListNode|} < 4/3$$

However, for the implementations considered, such a relation does not hold. Recall that, a node in the Simple Trie contains an integer type, a reference to a standard java Linked List for

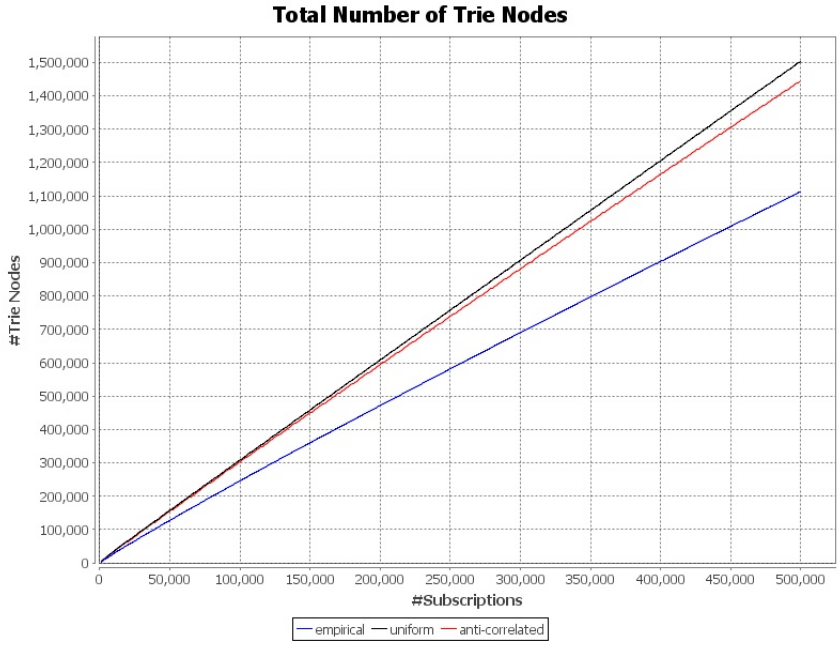


Figure 4.3: Vocabulary distribution impact on number of Trie nodes: $|\mathcal{S}| = 500,000$, $|\mathcal{V}_S| = 10,000$, $|S_i|_{AVG} = 4$

storing subscriptions and a reference to a HashMap for storing child nodes. Clearly this is much greater than a single Linked List node.

Trie - Term Occurences

Next, we capture the distribution of the number of Trie nodes for the three term frequency distributions in subscriptions. As Figure 4.4 depicts, when the empirical distribution is considered, the number of Trie terms rapidly increases, peaks and then decreases along with term rank. The peak in the first part of the plot is explained by the fact that the occurance of a term of the Trie exponentially increases for high ranked terms. Recall that by construction, it is allowed for low ranked terms to exist many times in the Trie. However, due to the empirical distribution, not many subscriptions containing such terms exist.

As Figure 4.5 illustrates, when a uniform distribution is considered, the number of times a term appears in the Trie monotonically increases as the ranks of the terms decrease. The frequency of the terms of the Trie corresponding to lower ranks is roughly bounded by 200. The interesting finding here, is that this upper bound for the Trie, is in accordance to the size of postings lists of the count based index as illustrated in Figure 4.2(b). When considering lower ranked terms, we can say that in a way the *Trie index degenerates to the Inverted File* with respect to the number of nodes both cases require per term (subscription posting nodes versus Trie nodes). The reason why this effect appears only in lower ranked terms is once again explained by the sharing of common prefixes. To better demonstrate consider the following example. Suppose two subscriptions $S_1 = \{t_{10000}, t_{9999}\}$ $S_2 = \{t_{10000}, t_{9998}\}$ are indexed by the Trie. Despite sharing a common subset ($\{t_{10000}\}$), since $\{t_{10000}\}$ is not a prefix, 4 terms would exist in the Trie (terms t_{10000},t_{9998}, and t_{9999} would exist 2, 1 and 1 times respectively). Observe that the count based approach would require the same number subscription postings when indexing S_1 and S_2. This effect does not apply however to higher ranked terms, since

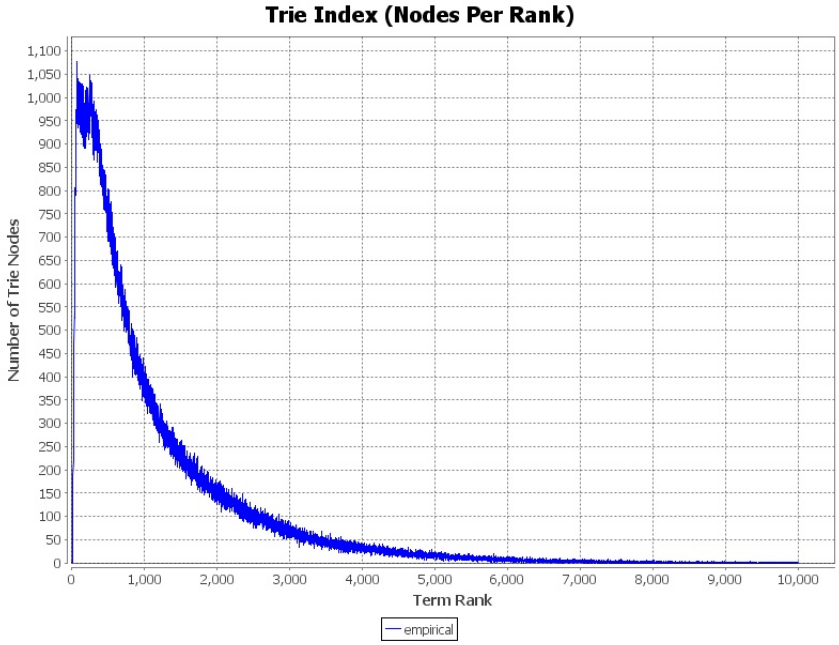


Figure 4.4: Term occurences per rank (empirical vocabulary distribution): $|\mathcal{S}| = 500,000$, $|\mathcal{V}_S| = 10,000$, $|S_i|_{AVG} = 4$

there is a higher probability for prefix sharing. In example, suppose two subscriptions sharing term t_{100}. Since uniform sampling is applied, there is a great probability that all other terms of both subscriptions, succeed term t_{100} (t_{100} would be a prefix). Hence, term t_{100}, as a prefix, would exist only once within the Trie.

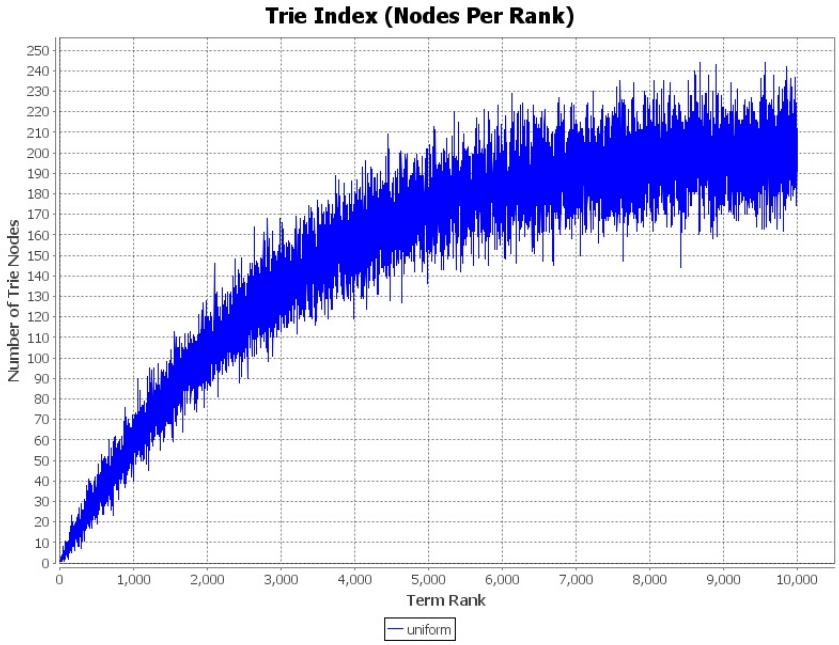


Figure 4.5: Term occurences per rank (uniform vocabulary distribution): $|\mathcal{S}| = 500,000$, $|\mathcal{V}_S| = 10,000$, $|S_i|_{AVG} = 4$

Finally, Figure 4.6 illustrates the corresponding Trie nodes when the anti-correlated distribution is considered in subscriptions. Once again, we can observe that the Trie index degenerates to the inverted file with regard to the number nodes assigned to same term but this time for terms with low ranks.

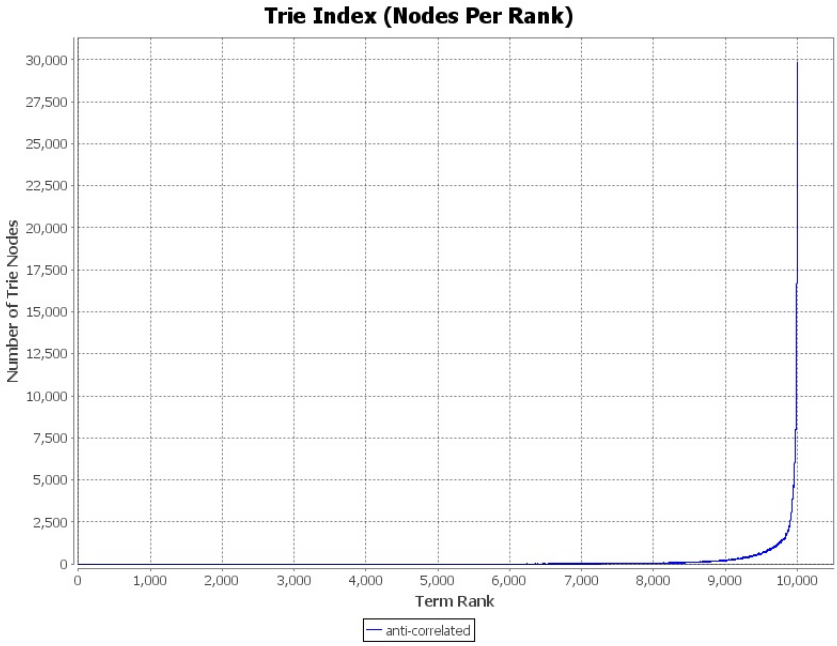


Figure 4.6: Term occurences per rank (anti-correlated vocabulary distribution): $|\mathcal{S}| = 500,000$, $|\mathcal{V}_S| = 10,000$, $|S_i|_{AVG} = 4$

Trie - Morphology

In this experiment we investigate the impact of the different subscription term distributions to the out-degree of the Trie nodes as well as the number of subscriptions assigned to them. Tables 4.2 and 4.3 illustrate our experimental findings.

As Table 4.2 illustrates that the vast majority of nodes in the three vocabulary distributions have up to three children. The percentage of leaf nodes (nodes with no out-degree equal to 0) for the empirical case is equal to 41%. For the uniform and anti-correlated case the total number of leaves is equal to 499,040 (or 33.2%) and 466,215 or (32.1%) respectively. In turn, the total number of internal nodes with a single child is equal to 617,964 (or 56.1%), 991,362 (or 66.1%), and 946,767 (or 65.2%) for the empirical, uniform, and anti-correlated distribution respectively. Observe that both the anti-correlated and uniform distributions behave in a similar manner with regard to the number of nodes containing a single child (out-degree equal to 1). When the anti-correlated and uniform case is considered, we do not anticipate many subscriptions to share a common prefix. Hence, the Trie would contain many single paths, which implies a great number of nodes containing only a unique child. The node with the highest number of children in all three cases is the root.

Table 4.3 presents the corresponding distribution of subscriptions over the nodes of the Trie. Two are the most significant findings. First, over 50% of the total number of nodes are not featuring any subscription for the three vocabulary distributions. More specifically, when the empirical case is considered a total of 649,305 nodes (i.e., 59%) of the Trie index are un-occupied.

Table 4.2: Out-degree distribution of nodes for Simple Trie index: $|\mathcal{S}| = 500,000$, $|\mathcal{V}| = 10,000$, $|S_i|_{AVG} = 4$

Out Degree	empirical	uniform	anti-correlated
0	454,837	499,040	466,215
1	617,964	991,362	946,767
2 - 3	37,854	5,062	24,875
4 - 7	11,710	1,034	1,313
8 - 15	4,740	1,269	926
16 - 31	2,207	1,522	1,055
32 - 63	1,132	1,700	1,083
64 - 127	626	1,795	1,400
128 - 255	294	1,209	1,370
256 - 511	118	0	0
512 - 1,023	67	0	0
1,024 - 2,047	15	0	0
2,048 - 4,095	1	0	0
8,192 - 16,384	0	1	1

The uniform and anti-correlated case, 'waste' 1,004,123 (i.e., 66.9%) and 963,006 (i.e., 66.4%) nodes respectively. Secondly, the anti-correlated case 'borrows' characteristics from both the uniform and empirical distribution. In terms of un-occupied nodes both the uniform and anti-correlated distribution behave in a similar manner. The total number of nodes that do not contain any subscription are in both cases close to 1,000,000. However, when considering the distribution of subscriptions for nodes with at least one subscription, the ant-correlated case behaves in a similar way to the empirical case. In both cases approximately 472,000 nodes have only a unique subscription.

Table 4.3: Distribution of subscriptions over the nodes of the Simple Trie: $|\mathcal{S}| = 500,000$, $|\mathcal{V}| = 10,000$, $|S_i|_{AVG} = 4$

Number of Subscriptions	empirical	uniform	anti-correlated
0	649,305	1,004,123	963,066
1	472,787	499,764	472,356
2 - 3	7,904	107	8,011
4 - 7	1,269	0	1,247
8 - 15	240	0	267
16 - 31	59	0	55
32 - 63	1	0	3

The awkward behavior exhibited by the anti-correlated case can be explained as follows. Recall that the anti-coorelated distribution is the exact inverse of the empirical one. Hence, the number of subscriptions, containing exactly the same terms is both distributions is the same. Their only differance lies in the fact that skewines is exhibited in the emprical for high ranked terms (versus low ranked for the anti-correlated case). For this reason no factorization oppurtunities are exhibited in the anti-correlated case, and thus the number of nodes without subscriptions is close to the uniform case.

The above experimental findings have motivated the compact versions of the two indices as presented in Sections 3.1.2 and 3.2.2.

4.1.2 Full Scale Evaluation

Next, we investigate the performance of the optimized indexes with regard to all three different distributions under a full scale evaluation. Work load parameters for this specific experiment were fixed to:

- $|\mathcal{S}| = 10,000,000$
- $|\mathcal{V}_S| = 800,000$
- $|I_k|_{AVG} = empirical$

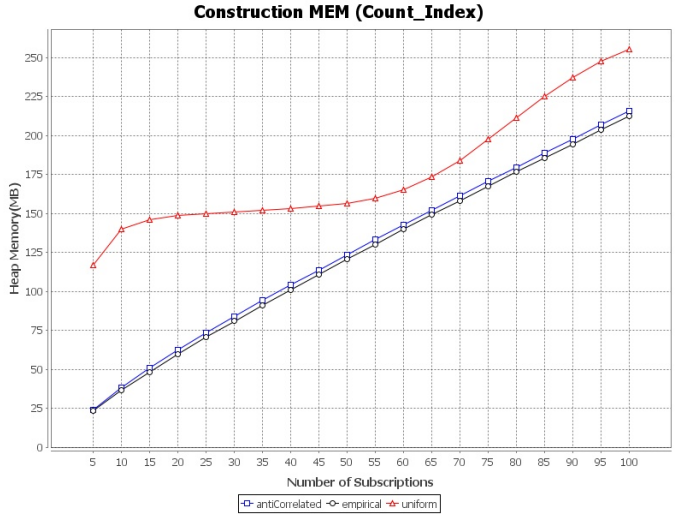
Count based index

As Table 4.4 illustrates, for the Count based index, both the empirical and anti-correlated case exhibit the same structural characteristics. Maximum hash table utilization is achieved when subscription vocabulary distribution is uniform (100% versus $\approx$ 73% for empirical and anti-correlated). This can be explained as follows. Within current implementation, the size of the hash table is initialized and remains fixed to $|\mathcal{V}_S|/2$ (400,000 in our case). When applying a uniform sampling scheme upon $|\mathcal{V}_S|$, given the large number of subscriptions, it is expected for the generated subscription set to cover a great (or whole in this case) part of the vocabulary. If this is the case then, the total number of terms found within the actual generated set of subscriptions is $> |\mathcal{V}_S|/2$. Hence, an entry exists for every hash table bucket (maximum capacity).

Table 4.4: Compact Count index characteristics when varying vocabulary distribution

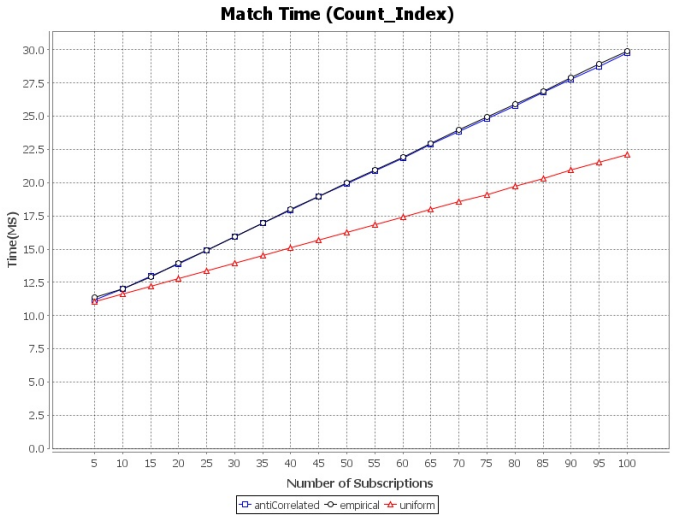
	empirical	**anti-correlated**	**uniform**
Total Hash Table Entries	292,457	292,480	400,000
Total Term Nodes	345,299	345,650	800,000
Average Collision List Size	1.18	1.182	2.00
Total Subscription List Nodes	1,308,676	1,308,965	1,501,159
Total Empty Slots	7,155,958	7,166,796	11,965,682
Average Empty Slots	20.72	20.73	19.02

As Figure 4.7(a) depicts, both the empirical and anti-correlated behave the same, with regard to the total number of memory they require ($\approx 210MB$). This is in accordance to the structural findings illustrated in Table 4.4. Interestingly, the memory requirements for the count based index when the uniform distribution is considered, exceeds by 20% the other two cases. This is due to implementation, and results from the fact that we consider a compact array list data structure for storing subscription postings. According to the results of Table 4.4, when the uniform case is considered, the count based index requires 192,483 (or 14.7%) more posting list nodes than the other two cases. In addition, the index contains in total 4,809,697 (or $\approx$ 67.1%) more empty slots.



(a) Memory requirements

(b) Build time requirements



(c) Matching time requirements

Figure 4.7: Vocabulary distribution impact on Compact Count based index: $|\mathcal{S}| = 10,000,000$, $|\mathcal{V}_S| = 800,000$, $|I_k|_{AVG} = empirical$

According to Figure 4.7(b), the time required to index a subscriptions when the uniform case is considered is greater than the other two cases, but the difference is not significant. As expected, for all three distributions the required time is independent to the number of currently indexed subscriptions. Finally, as Figure 4.7(a) illustrates, both the uniform and anti-correlated case achieve approximately the same matching performance. The best performance is achieved when both subscriptions and items follow the uniform distribution ($\approx 33\%$ gain when compared to empirical and anti-correlated distribution). When considering an uniform sampling scheme for subscriptions, it is anticipated for subscriptions to be evenly distributed over the vocabulary. This even partitioning of the subscription search space, results to fewer subscription posting traversals when compared to the other two distribution cases. Recall, that both items and subscriptions follow the same distribution. The news item throughput rate for the worst case is equal to 33 items/sec.

Trie index

As illustrated in Table 4.5, which depicts the findings for the Trie index, the total nodes occupied by the empirical, uniform and anti-correlated distribution case is equal to 7,419,246 , 7,397,159 and 8,635,334 respectively. Interestingly, the variation in all three cases is less than 16% (anti-correlated and empirical only vary by 0.3%). This contradistincts the findings of our previous experiment depict in Figure 4.4, where a maximum variation of 33% was observed. This behaviour results from the fact that, as opposed to the previous experiment where the simple trie was considered, the Trie evaluated in this experiment, considers path compression. Many nodes corresponding to single paths, are represented in the compressed Trie as a single compact node. This is also verified by the results obtained. As Table 4.5 illustrates the anti-correlated and uniform case occupy 14.78% and 11.97% more compact nodes than the empirical case. Clearly, the impact of the distribution on the total number of nodes is confiscated by path compression.

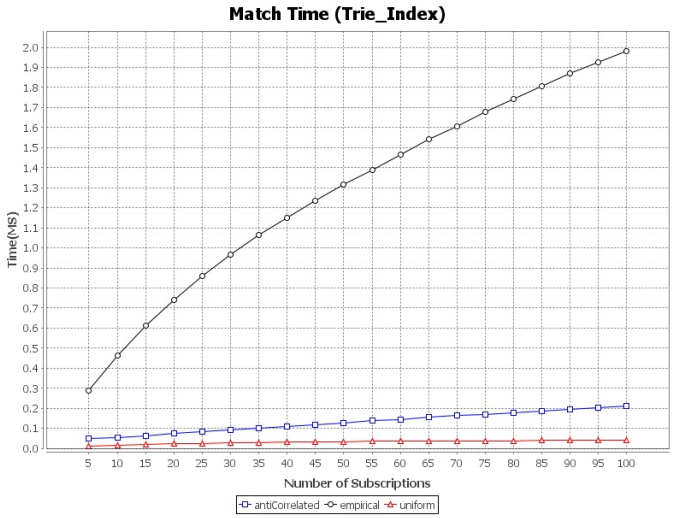
Table 4.5: Compact Trie characteristics when varying vocabulary distribution

	empirical	**anti-correlated**	**uniform**
Total Nodes	7,419,246	7,397,159	8,635,334
Total paths(leafs)	6,587,124	6,791,385	7,914,044
Average path length	3.456	3.064	2.990
Internal nodes percentage	11.22%	8.19%	8.35%
Leaf nodes percentage	88.78%	91.81%	91.65%
Total compact nodes	2,707,485 (36.49%)	3,792,186 (51.27%)	4,184,549 (48.46%)
Average compact node size[1]	2.687	2.90	2.845

The results obtained for the memory requirements of the Trie index are illustrated in Figure 4.8(a). As we can see, the empirical case performs the best space wise as it requires approximately 625MB. Interestingly, the anti-correlated case and uniform perform quite well. The uniform case needs 16% more memory than the empirical case; when an anti-correlated distribution is considered the memory requirements for the index slightly increase about roughly 4%. The results are in accordance to the structural findings showed in Table 4.5. In addition, as illustrated in Figure 4.8(b), the time required to index a subscription, is as expected, independent to subscription vocabulary distribution and the number of subscriptions currently indexed. This verifies our analytical model devised in earlier chapters.

(a) Memory requirements

(b) Build time requirements



(c) Matching time requirements

Figure 4.8: Vocabulary distribution impact on Compact Trie: $|\mathcal{S}| = 10,000,000$, $|\mathcal{V}_S| = 800,000$, $|I_k|_{AVG} = empirical$

The matching performance for all three distribution cases is illustrated in Figure 4.8(c). Surprisingly, when both subscriptions and items follow the empirical case, the achieved performance is 10 times greater than the anti-correlated (and uniform) case. This can be explained as follows. As opposed to the anti-correlated (and uniform) vocabulary distribution case, when an empirical distribution is considered, it is expected for many subscriptions to share a common prefix. The Trie index would therefore contain more internal nodes that are in addition not compact. Hence, when matching, the algorithm would need to recursively visit more Trie nodes. This does not hold for the other two cases though, since in order for the matching algorithm to proceed to a particular compact node, all of the terms of that node must also be present in the item being matched. Therefore, in total, fewer nodes will be visited. Also observe that the matching is sub-linear in the number of subscriptions. This is more obvious for anti-correlated and uniform case. In the worst case, the Trie index achieves a throughput rate of 500 items/sec.

Vocabulary Distribution impact on Matching

In this SubSection we study the impact of the term frequency distribution to the number of operations needed for matching. More specifically, for the count based index we measured the times a counter decrement is performed; for the Trie index we captured the number of nodes visited. The workload parameters for this experiment were fixed to:

- $|\mathcal{S}| = 10,000,000$
- $|\mathcal{V}_S| = 800,000$
- $|S_i|_{AVG} = empirical$
- $|I_k|_{AVG} = empirical$

To obtain the credibility of our measurements, matching performance was averaged over a set of 10,000 different items generated according to the workload parameters described above. Table 4.6 depicts the results.

Table 4.6: Matching Operations of Trie and Count-based Indices

Subscription Term Dist	**Item Term Dist**	**# Operations**	
		Inverted File	**Trie**
empirical	empirical	215,620.17	260.99
	uniform	544.12	3.28
anti-correlated	anti-correlated	210,106.34	211.56
	uniform	544.64	5.39
uniform	empirical	591.67	19.46
	anti-correlated	576.82	17.78
	uniform	559.62	18.31

The worst case performance for the inverted file is exhibited when both the items and subscriptions follow the same distribution. The total amount of operations required is on average 215,620.17 or approximately 396.7 times more operations than in the case when both subscriptions and items follow the empirical/uniform distribution. Recall that, the size of a posting list in the Inverted File (IF) is proportional to the frequency of its corresponding term. If both items and subscriptions follow the same skewed distribution then, the number of subscription postings examined (which is equal to the number of the performed counter decrements) will be large. Observe that the Trie index requires significantly less operations upon matching, when

compared to the IF. In the worst case, where subscription and item terms follow the empirical distribution, the matching algorithm of the Trie index requires on average to examine 260.99 nodes. The Trie outperforms the inverted file index in all cases considered.

4.2 Impact of the Subscription Size

In this SubSection, we study the impact of the subscription size on the two indexes. We consider four subscription size cases: $|S_i|_{AVG} \in \{3, 6, 9, 12\}$. Other workload parameters are fixed and set to:

- $|\mathcal{S}| = 10,000,000$
- $|\mathcal{V}_S| = 800,000$
- Subscription term distribution: empirical
- Item term distribution: empirical
- $|I_k|_{AVG} = empirical$

As predicted by our analytical cost model of the Count-based index the number of nodes in subscription postings increase along with the size of subscriptions. In Table 4.7 we can see that a fourfold increase in subscription size, results to a 2.5 times increase in the number of posting nodes of the Inverted File. Furthermore, larger subscriptions imply a greater covering of the vocabulary $\mathcal{V}_S$. According to Table 4.7, the total number Hash table entries (i.e., terms) increase along with $|S_i|_{AVG}$. We can observe that the average number of empty slots of the custom array based list data structure we employed for implementing subscription postings is independent to the subscription size. As depicted in Figure 4.9(a), the memory requirements of IF, when $|S_i|_{AVG} = 12$ are 750MB. A fourfold increase in the size of subscriptions results to a 1.9 (450MB) times increase in memory. This kind of behavior verifies the structural characteristics of the IF presented in Table 4.7.

Table 4.7: Compact Count index characteristics for varying Subscription Sizes

	$\mathbf{\lvert S_i \rvert_{AVG} = 3}$	$\mathbf{\lvert S_i \rvert_{AVG} = 6}$	$\mathbf{\lvert S_i \rvert_{AVG} = 9}$	$\mathbf{\lvert S_i \rvert_{AVG} = 12}$
Total Hash Table Entries	307,370	363,287	383,922	392,692
Total Term Nodes	371,651	496,922	575,226	629,374
Average Collision List Size	1.209	1.368	1.498	1.603
Total Subscription List Nodes	1,507,916	2,806,711	4,064,412	5,300,799
Total Empty Slots	7,697,900	10,169,275	11,610,300	12,519,975
Average Empty Slots	20.71	20.46	20.18	19.89

As we can se in Table 4.8 increasing the size of subscriptions does not significantly affect the total number of nodes that the Trie needs to construct (less than 3% variation). This behavior can be explained as follows. Given the large size of the vocabulary it is quite rare that subscriptions will share many common terms. This effect become even more intense as the size of subscriptions also increases and it is reflected to the Trie structure by the increasing number of compact nodes that may exist (i.e., nodes compressing paths with a unique child). Hence, increasing the size of subscriptions does not imply the creation of a large amount of nodes but to compact nodes assigned to more terms. Returning to Table 4.8 when $|S_i|_{AVG} = 3$ the total number of compact nodes is equal to 27.33%; when $|S_i|_{AVG} = 6$, then the total number of

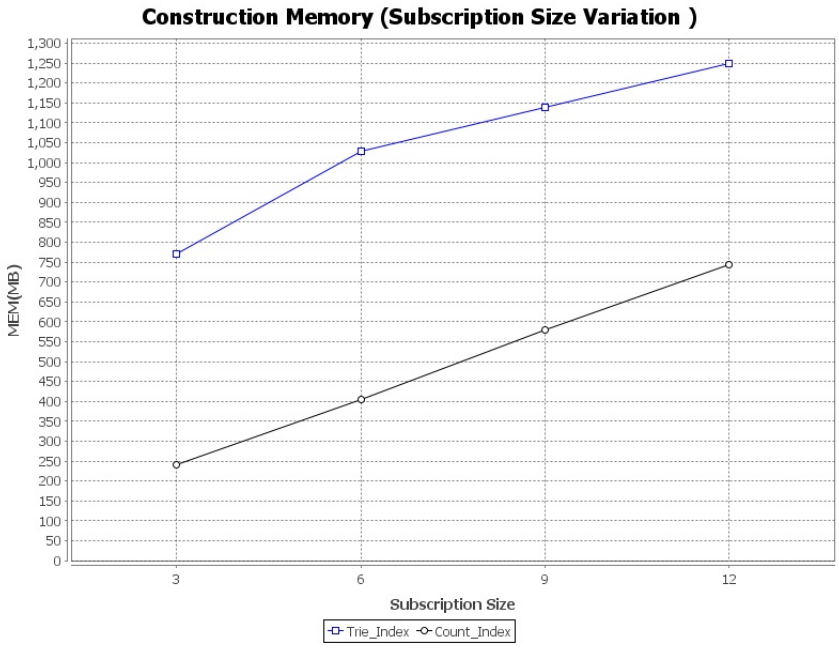
compact dramatically increases to 88.91%. In addition, we can observe that when $|S_i|_{AVG} > 3$, every subscription is stored in a different leaf node (recall that $|\mathcal{S}| = 10,000,000$); this means that no identical subscriptions exist. This is yet another indication that larger subscription sizes result only to larger compact nodes.

Table 4.8: Compact Trie Characteristics for Varying Subscription Sizes

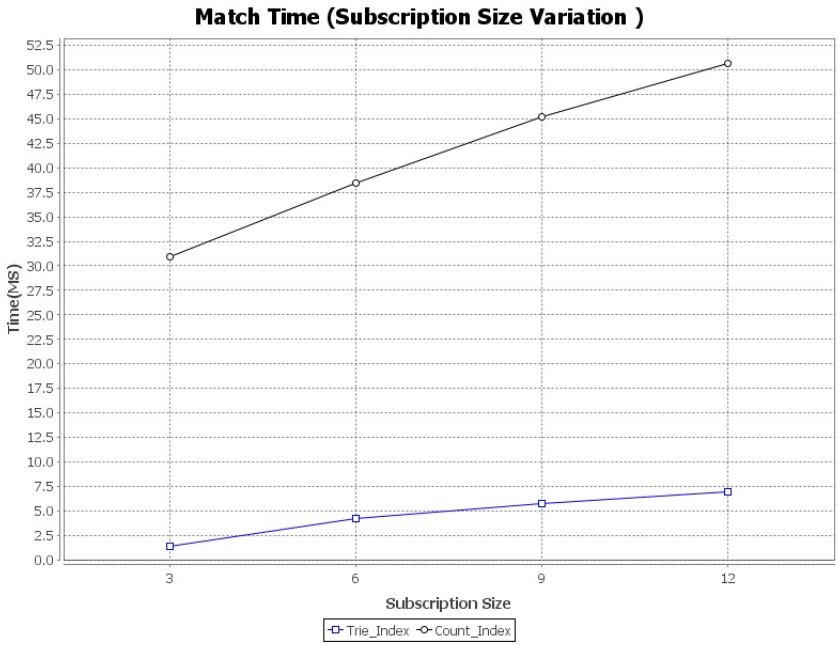
	$\|\mathbf{S_i}\|_{\mathbf{AVG}} = \mathbf{3}$	$\|\mathbf{S_i}\|_{\mathbf{AVG}} = \mathbf{6}$	$\|\mathbf{S_i}\|_{\mathbf{AVG}} = \mathbf{9}$	$\|\mathbf{S_i}\|_{\mathbf{AVG}} = \mathbf{12}$
Total Nodes	11,247,416	11,247,290	11,507,983	11,652,566
Total paths(leafs)	9,972,546	10,000,000	10,000,000	10,000,000
Average path length	3.691	4.081	4.387	4.641
Internal nodes percentage	11.33%	11.09%	13.10%	14.18%
Leaf nodes percentage	88.67%	88.91%	86.90%	85.82%
Total compact nodes	3,073,859 (27.33%)	10,000,398 (88.91%)	10,001,269 (85.91%)	10,002,386 (85.84%)
Average compact node size	2.001	3.917	6.611	9.356

As we can see in Figure 4.9(a), the memory requirements for the Trie index when considering $|S_i|_{AVG} = 12$ is equal to 1250MB (66% greater than the count based index). A fourfold increase in subscription size results to a 56.25% increase in memory. As depicted in Figure 4.9(a), for subscription sizes greater than six, we observe a linear memory increase. This stems from the fact that, for subscription sizes greater than six and large vocabularies (in this experiment $|\mathcal{V}|_{AVG}$), each individual subscription is stored a different leaf node.

Finally, as we can see in Figure 4.9(b), with regard to matching time, the Trie outperforms the Count index by an order of magnitude. Recall that according to the analytical cost of Equation 2.9, the matching time for the Count-based index increases linearly with respect to the subscription size. Quite surprisingly this linear correlation is also observed for the Trie index and can be explained as follows. The Trie matching time depends to the number of nodes actually traversed by the matching algorithm. For subscription sizes greater than 6 that we observed a linear memory increase, matching time also increases linearly.



(a) Build memory of Compact Trie and Compact Count index



(b) Matching ime of Compact Trie and Compact Count index

Figure 4.9: Subscription Size impact on Compact Trie and Compact Count indexes: $|\mathcal{S}| = 10,000,000$, subscription vocabulary distribution: empirical, item vocabulary distribution: empirical , $|I_k| = empirical$, $|\mathcal{V}_S| = 800,000$

4.3 Impact of the Vocabulary Size

In this SubSection, we are interested in studying the effect on the two indices of an increasing vocabulary size. We consider three distinct values, $\mathcal{V}_S = 10,000$, $\mathcal{V}_S = 100,000$ and $\mathcal{V}_S = 800,000$ upon which synthetic data generation will be applied. Other work load parameters are fixed and set to:

- $|\mathcal{S}| = 10,000,000$
- Subscription term distribution: empirical
- Item term distribution: uniform
- $|I_k|_{AVG} = empirical$
- $|S_i|_{AVG} = empirical$

As we can see from Table 4.9, Hash table utilization decreases as vocabulary size[2] increases. In addition, the number of empty slots assigned to the last element of the subscription posting set of each term, also increase. This can be explained as follows. Bigger vocabulary sizes results to a sparser distribution of subscriptions over the terms of the vocabulary. As a result, a larger number of entries are created in the Hash table which in turn implies more posting set elements with empty slots. As expected, we do not observe any significant variation in the size of subscription posting sets. Table 4.10 illustrates the corresponding behavior of the Trie index. As we have seen in Section 2.3, the Trie optimizes the space requirements by factorizing common prefixes and assigning them to common paths. Obviously, the number of common prefixes that subscriptions would share is reversely proportional to the size of the vocabulary. This justifies the increase of the total number of Trie nodes illustrated in Table 4.10, as well as, the increase in the total number of compact nodes.

Table 4.9: Compact Count index characteristics for varying Vocabulary Size

	$\|\mathcal{V}_\mathbf{S}\| = \mathbf{10,000}$	$\|\mathcal{V}_\mathbf{S}\| = \mathbf{100,000}$	$\|\mathcal{V}_\mathbf{S}\| = \mathbf{800,000}$
Total Hash Table Entries	5,000	49,904	292,457
Total Term Nodes	10,000	90,248	345,299
Average Collision List Size	2.00	1.808	1.18
Total Subscription List Nodes	1,026,909	1,091,189	1,308,676
Total Empty Slots	109,187	1,716,793	7,155,958
Average Empty Slots	10.92	19.02	20.72

As depicted in Figure 4.10(a), the Count index clearly requires four times less memory than the Trie. As expected, the memory required by the IF increases linearly with respect to the size of the vocabulary[3]. Increased vocabulary sizes also affect the memory requirements of the Trie. More precisely, an eightfold increase of the vocabulary results to a 75MB(50%) and 350MB(93%) increase for the IF and Trie respectively. It should be stressed that compared to the effect of increasing subscription sizes (see Figure 4.9) the impact of vocabulary size in the IF is significantly smaller (50% versus 190%).

Figures 4.10(b) and 4.10(c) depict respectively how matching time of Count and Trie vary with respect to the vocabulary size. In both indices, increased vocabulary sizes results to a

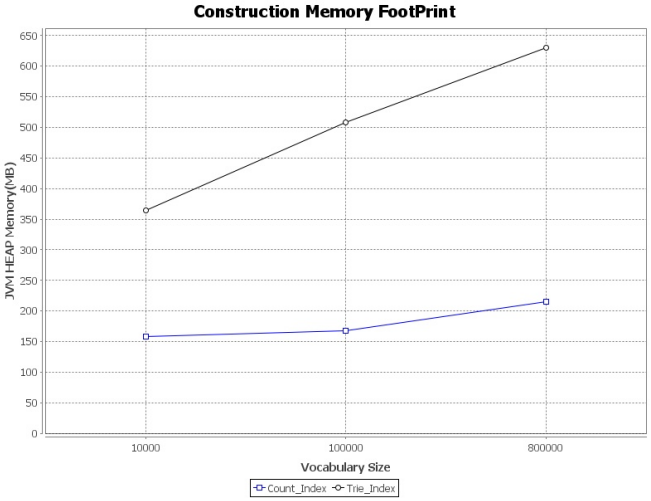
[2]Unless otherwise specified, we will use the phrase 'vocabulary size' to refer to the size of the 'universal' vocabulary upon which generation is applied

[3]Please note that in Figure 4.10(a) it does not appear to be linear because the 'x' axis is not in normal scale

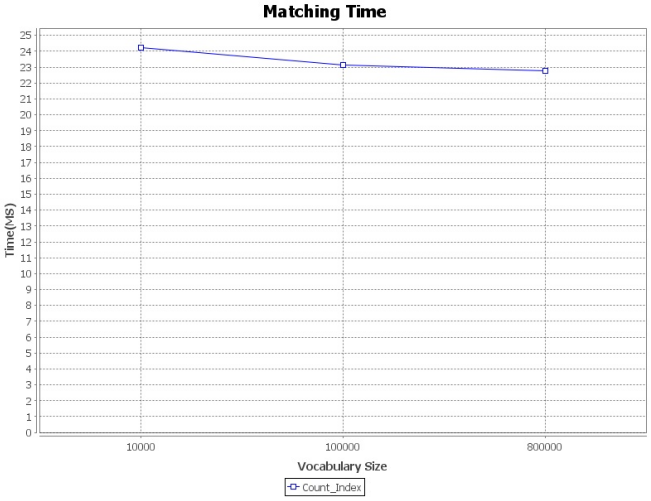
Table 4.10: Compact Trie characteristics for varying Vocabulary Size

	$\|\mathcal{V}_\mathbf{S}\| = \mathbf{10,000}$	$\|\mathcal{V}_\mathbf{S}\| = \mathbf{100,000}$	$\|\mathcal{V}_\mathbf{S}\| = \mathbf{800,000}$
Total Nodes	2,970,398	5,399,669	7,419,246
Total paths(leafs)	2,553,389	4,754,301	6,587,124
Average path length	4.788	3.979	3.465
Internal nodes percentage	14.04%	11.95%	11.22%
Leaf nodes percentage	85.96%	88.05%	88.78%
Total compact nodes	645,138 (21.72%)	1,653,656 (30.63%)	2,707,485 (36.49%)
Average compact node size	2.457	2.601	2.687

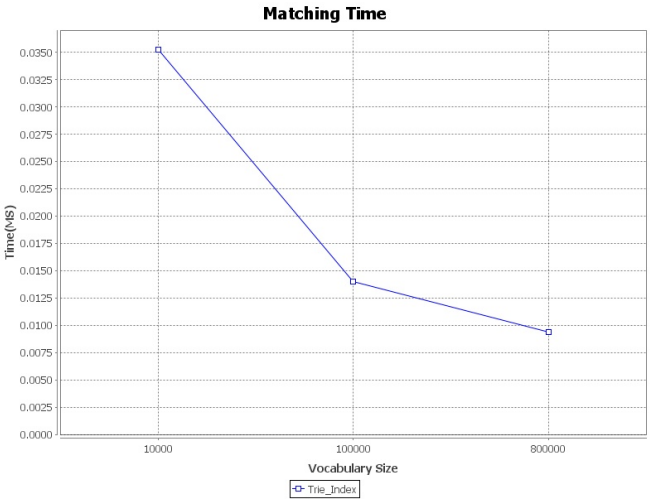
decreasing time for matching an incoming news item against the set of indexed subscriptions (for the Count index see Equation 2.9). We can observe however that the decrease in matching time for the Trie index becomes more important. Trie matching performance improves as bigger vocabulary sizes implies a sparser distribution of subscriptions to the vocabulary terms. At this point recall that, both index structures partition the set of subscriptions according to the terms they contain; the IF considers a flat partitioning scheme while the Trie a hierarchal one. Hence, the matching performance of the Trie benefits from bigger vocabularies since a fine-grained search space is actually explored.



(a) Memory Requirements of Compact Trie and Compact Count



(b) Compact Count index - Matching



(c) Compact Trie index - Matching

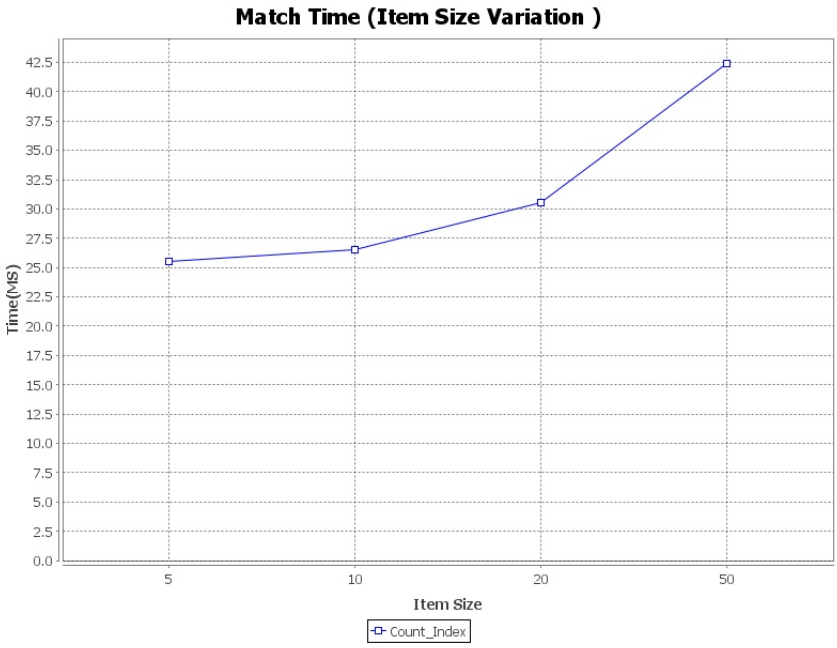
Figure 4.10: Vocabulary size impact: $|\mathcal{S}| = 10,000,000$, Subscription term distribution: empirical , Item term distribution: uniform , $|I_k|_{AVG} = empirical$, $|S_i|_{AVG} = empirical$

4.4 Impact of the News Item Size

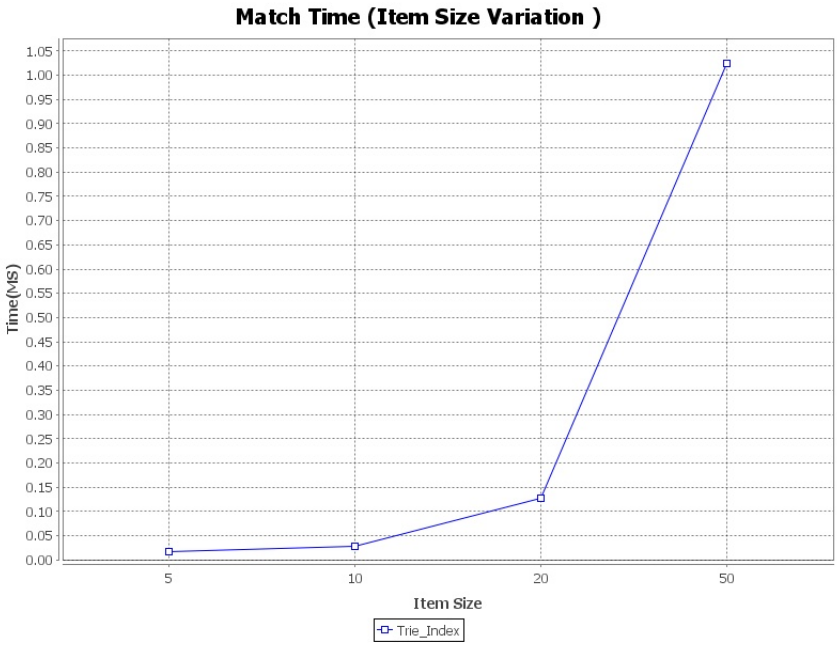
In this SubSection, we study the effect of varying the size of incoming news items. We consider the following set of values for $|I_k|_{AVG} \in \{5, 10, 20, 50\}$ and capture matching performance in each case. All other workload parameters are set to:

- $|\mathcal{S}| = 10,000,000$
- $|\mathcal{V}_S| = 800,000$
- Subscription term distribution: empirical
- Item term distribution: empirical
- $|S_i|_{AVG} = empirical$

Figure 4.11(a) illustrates the results for the Count based index. According to the complexities presented in Table 2.3, the matching time of Count index increases linearly with respect to the size of news items. Recall, that for every term within the matching item, the inverted set of subscription postings is obtained and traversed. Greater item sizes, result to more subscription list traversal. As we can see in Figure 4.11(a), a tenfold increase in the subscription size leads to a 68% increase in the time required for matching. However, this effect because more intense in the Trie. As we can see in Figure 4.11(b), the time required to match an incoming news item against 10,000,000 indexed subscriptions increases exponentially along with its size. This exponential increase can be explained as follows: the time spent when matching for the Trie index is proportional to the number of nodes the algorithm traverse. The algorithm traverse a total number of paths, equal to all possible subsets of the terms of the item which are also present in the indexed subscriptions. Hence, larger item sizes imply on average to traverse more paths in the Trie.



(a) Compact Count index



(b) Compact Trie index

Figure 4.11: Item size impact on Compact Trie and Compact Count indexes: $|\mathcal{S}| = 10,000,000$, $|\mathcal{V}_S| = 800,000$, Subscription term distribution: *empirical*, Item term distribution: *empirical* , $|S_i|_{AVG} = empirical$

4.5 Evaluation on Scalability

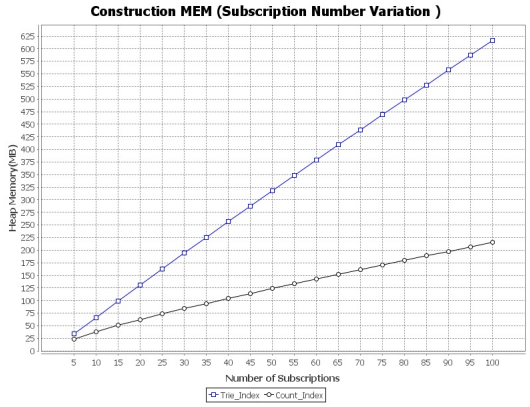
In this final experiment we evaluated the scalability of both indices. More specifically, we incrementally indexed a total of $10,000,000$ subscriptions. At each step, a batch of $500,000$ subscriptions were added to the index under evaluation. After loading each batch, we measured the memory occupied by the indices, and the time required to match an individual news item. Additionally, we captured the time required to index an individual subscription of a specific batch (averaged over the set of subscriptions of each batch).

The workload paramters were set to:

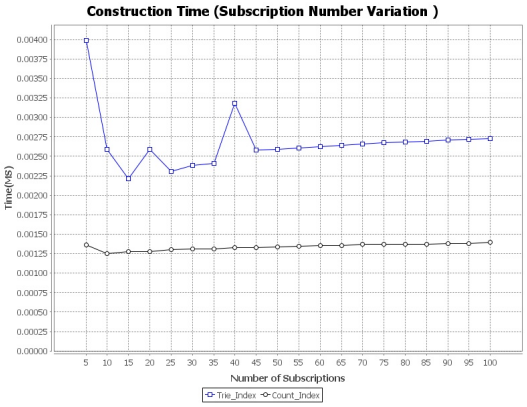
- $|\mathcal{S}| = 10,000,000$
- $|\mathcal{V}_S| = 800,000$
- Subscription term distribution: empirical
- Item term distribution: empirical
- $|S_i|_{AVG} = empirical$
- $|I_k|_{AVG} = empirical$

As Figure 4.12(a) illustrates, when indexing a total of $10,000,000$ subscriptions the Count and Trie index require approximately $215MB$ and $625MB$ respectively (the Count index outperforms the Trie by 1.9 times). As expected (see memory model in 2.8) the Count index exhibits a linear behaviour with regard to the number of subscriptions indexed. Quite surprisingly, this is also observed for the Trie. We would expect that as the number of subscriptions increase, the Trie would occupy more nodes (representing the prefix relations of the set of indexed subscriptions), hence there would be a greater probability that indexing new subscriptions would require less space as there is a greater probability that they would share common prefixes of subscriptions already indexed in the Trie; i.e. in a best case scenario, indexing a subscription S_i, would result to only storing S_i to an already existing node (this requires that there exists a subscription S_j s.t. $S_j \succeq S_i$). However, given the large vocabulary size, it is clear that even when indexing a total of $10,000,000$ subscriptions the anticipated saving due to prefix sharing is not significant.

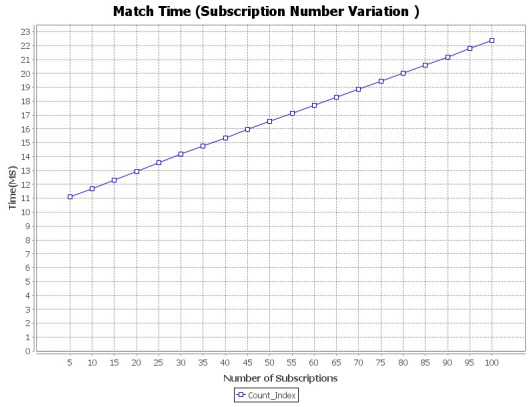
According to Figure 4.12(b), the time required to index an individual subscription for the Count index is slightly less than that of the Trie. For both indices, the time required is independent to the number of subscriptions currently indexed. This is in accordance to the analytical models (see 2.1 and 2.13 respectively) devised in previous chapters. Finally, as we can see in Figures 4.12(c) and 4.12(d), the Trie outperforms the Count index by 2 orders of magnitude (22MS versus 0.01MS) when matching. As opposed to the Trie, for which the matching time appears to stabilize as the number of indexed subscriptions increase, the time required when matching for the Count index increases linearly along with the number of indexed subscriptions.



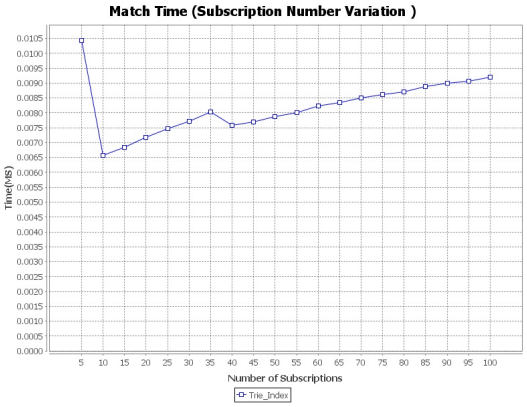
(a) Memory requirements of both indices



(b) Time required to index a subscription



(c) Matching time for Compact Count index



(d) Matching time Compact Trie index

Figure 4.12: Scalability characteristics of Compact Trie and Compact Count indices: $|\mathcal{S}| = 10,000,000$, $|\mathcal{V}_S| = 800,000$, Subscription term distribution: *empirical*, Item term distribution: *empirical*, $|S_i|_{AVG} = empirical$, $|I_k|_{AVG} = empirical$

4.6 Summary on experimental results

The main conclusions drawn from our experiments regarding the impact of critical workload parameters to the *morphology* of the two indices are:

Vocabulary Distribution: The number of subscription entries that need to be stored in the postings sets of the Count-based index is independent of the subscription vocabulary distribution. However, the actual distribution of the sizes of the various postings sets depends heavily on the terms' frequency distribution. As a matter of fact, more *frequent terms will be assigned to larger posting sets* which in turn directly impacts matching time. On the other hand, the total number of terms in the Trie is affected *by the subscription vocabulary distribution with respect to the initial ranking considered.* The most favoured case for the Trie is when the subscription vocabulary distribution does not violate the initial ranking considered for its construction. In this case, a great part of high ranked terms would appear in the Trie structure (i.e., left deep tree). In the opposite, a large number of low ranked terms would appear in the Trie (i.e., right width tree). Since prefix sharing will not be activated many single paths would exist and the Trie essentially degenerates to the inverted file. Although the occurrences of the terms depends on the underlying relation between the vocbulary distribution and the ranking considered by the Trie, the actual number of nodes does not vary significantly (empirical distribution requires $\approx 14\%$ more nodes), due to the path compression optimization employed for the Trie (uniform and anti-correlated distribution occupy roughly 15% more compact nodes). The *width and height of the Trie is independent of the vocabulary distribution.*

Subscription Size: The number of nodes in postings sets of the Count- based index increases linearly along with the size of subscriptions. On the other hand, the size of subscriptions impact the *height* of the Trie index: larger subscriptions imply deeper Trie structures. In addition, the percentage of compact nodes gets almost stabilized after a certain subscription size (percentage of compact nodes stabilized at roughly 86% for $|S_i| > 6$). This is due to the fact that each separate subscription is actually stored in a different leaf node and, thus, greater subscription sizes result to larger compact nodes i.e. factorizing more terms. More precisely the size of compact nodes increases linearly with respect to the size of subscriptions.

Vocabulary Size: The number of subscription entries in the postings sets of the Count- based index is independent of the vocabulary size. On the other hand, the size of the vocabulary impacts the *width* of the Trie index: bigger vocabularies imply broader Trie structures. The ratio of leaves to internal nodes remains almost stable ($< 3\%$ variation) due to an increase in the number of compact nodes (10% per scale of vocabulary size).

In this context, the main performance figures exhibited by the Trie and Count-based indices are:

Memory Requirements: In all cases, the *Count-based index outperforms the Trie memory wise.* In our experiments, their difference lies between almost of half more memory (for empirical distribution of vocabulary terms and subscription sizes, $|\mathcal{V}_\mathcal{S}| = 10,000$) and of a double memory (for empirical distribution of vocabulary terms and $|S_i| = 3, |\mathcal{V}_S| = 800,000$) required by the Trie w.r.t. the Count-based in order to index 10,000,000 subscriptions. As described in section 1.1.3 we expect in a realistic setting the vocabulary to be large (order of 10^6) and average subscription sizes to fall between 2-3 terms (as in web queries) and

4-5 terms (as in advertisement queries). Given that (a) the vocabulary impact is more outstanding for the Trie when compared to the Count-based index ($400MB$ increase versus $200MB$ increase for vocabulary size from $|\mathcal{V}_S| = 10,000$ to $|\mathcal{V}_S| = 800,000$), and (b) that larger subscription sizes similarly impact both indices ($500MB$ and $490MB$ increase in memory for a four fold increase in subscription size for the Count and Trie respectively), we anticipate that this realistic setting the Trie would require almost double memory than Count-based. When we consider the empirical distribution for the size and vocabulary of subscriptions, to index 10,000,00 subscriptions the Trie requires 1.99 times more memory than the Count index ($625MB$ versus $215MB$).

Matching Time: In all cases, the *Trie outperforms the Count index with respect to matching time*. In the best case (when the subscription vocabulary and size follow the empirical distribution, news items vocabulary follows the uniform distribution and $|\mathcal{V}_S| = 800,000$) the time required for matching an incoming news item against a set of 10,000,000 indexed subscriptions is 3 orders of magnitude greater than that of the Count index. Even in the worst case (when the subscription/news item vocabulary follows the empirical distribution and $|S_i| = 12$, $\mathcal{V}_S = 800,000$,) the matching time required by the Count-based index is still 1 order of magnitude greater than that of the Trie index. In a realistic setting with characteristics similar the workload previously described and in addition with an empirical news items size distribution the Trie outperforms by 1 order of magnitude ($2ms$ versus $30ms$) the Count index when matching a single news items against a set of 10,000,000 subscriptions.

Construction Time: The number of currently indexed subscriptions does not impact the build time of both indices. In addition it is not affected by the vocabulary distribution in subscriptions.

With respect to matching, the impact of the size of the news item is more outstanding for the Trie as opposed to the Count-based index (2 orders of magnitude versus 68% increase for a 10 fold increase of $|I_k|$). As predicted by the cost models of Chapter 2 (see Table 2.3 and Equation 2.21), the Count and Trie index exhibit a linear and exponential dependency, in the size of the news item being matched, correspondingly. However, even in the worst case ($|I_k| = 50$), which is close to the anticipated workload of a realistic setting, the Trie outperforms the Count-based index by an order of magnitude.

The memory and matching requirements for the Count based index w.r.t the number of indexed subscriptions *increase linearly*. For the Trie index, an increase in the number of subscriptions results to a linear increase in the memory required but a sub linear increase in the time required for matching. Hence, both subscription indices address the matching problem in a scalable manner.

As stated in Section 1.2, a Publish/Subscribe system should perform an on the fly news item processing to alleviate the need for storing incoming news items. If we consider the throughput rate as the number of news items the event processing system can process per second, then the underlying subscription index must guarantee a throughput rate for news items greater than their publishing rate. When subscription (and news item) size and vocabulary follow the empirical distribution then the Trie index loaded with $10,000,000$ subscriptions can guarantee a throughput rate of ≈ 500 $items/sec$ as opposed to the much smaller ≈ 34 $items/sec$ of the Count-based index at the expanse of 1.9 times more memory. Clearly, in a realistic setting (as described in Section 1.1.3), the Trie as opposed to the Count index can easily guarantee on the fly news item processing given (a) the high publishing rate ($> 10/h$) and burstyness (up to 215

times the average publication rate) that the news feeds can exhibit, and (b) the large amount of publishing sources

Chapter 5

Related Work

Their has been much interest in event processing systems. In such systems, users submit long lasting subscriptions which get evaluated over a stream of incoming events. Depending on target application domain, different workload characteristics may be applied. This disparity on application requirements has lead to the development of a great number of event processing systems with a great variety of expressiveness for subscriptions. Evidently, a trade off between subscription language expressiveness and evaluation performance does exist. Figure 5.1 depicts a rough classification scheme based upon the expressiveness provided by such systems in contradistinction to the number of subscriptions they can efficiently handle.

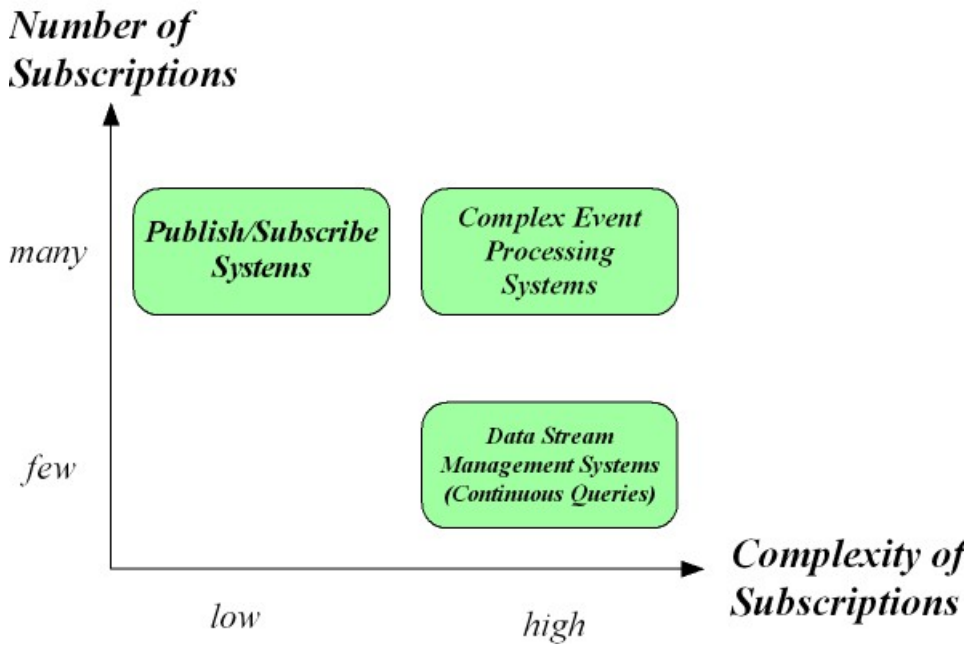


Figure 5.1: Event Processing systems

At the one end of the spectrum lie *Publish/Subscribe* systems. The goal of such systems is to be able to handle a great number of subscriptions while simultaneously achieving a high throughput rate for events. Hence, efficiency is attained at the expense of subscription language expressiveness. The earliest publish subscribe systems where topic based. Example such systems amongst others are Corona [36], and SCRIBE [38].

The notion of topics restricts the subscriber as he/she can only be notified about events

belonging to a particular group. Content based publish subscribe on the other hand provides a more fine grained subscription model. Such systems give the user additional flexibility on what events will he/she be finally notified about since subscriptions are now targeted on event content. Examples of such systems are GRYPHON [4], SIENA [12] and LeSubscribe [33].

At the other end of the spectrum lie *Data Stream Management Systems* (DSMS). A data stream management system, as opposed to that of traditional data base management systems , is based upon the notion of real time data streams. A data stream is considered as a sequence of objects. The order of the objects within that sequence could either be explicitly defined by the source or implicitly be given by the DSMS itself in the form of an unique identifier. Every object within the sequence consists of a set of attributes, in a similar manner to that of tuples of a relational data base. However, as opposed to traditional DBMS where tuples are persistently stored which permits data to be read more than once, the streaming nature of DSMS introduces the constraint that objects within a particular stream can only be read once. Queries submitted by users are long lasting(continuous), an analogy to subscriptions of publish subscribe systems. The query languages that these specific types of systems provide are based upon the notion of order and time and contain amongst others operators for selection, join, aggregation, multiplexing and others. The form of the queries, reflects the underlying data model (i.e. relational, object based, semi-structured, etc). An example of a relational based query language used within the STREAM system is CQL [31]. Another example of a system supporting continuous querying over streaming data is that of TelegraphCQ [13] system. Finally the work of [3] proposes the BOREALIS prototype DSMS that queries data based on procedural model. As opposed to Publish/Subscribe systems where only selection operations are applied in the form of subscription predicates, DSMSs target application domains where complex data stream querying is the need (i.e. stock quote analysis, sensor network monitoring etc). This additional expressiveness however comes at the expense of scalability in terms of the number of queries such systems can support.

Publish subscribe systems achieve high efficiency at the expense of subscription language expressiveness. Data stream management systems on the other hand provide continuous query languages at the expense of scalability. A great deal of work has been conducted recently to bridge the gap among the two in the form of *Complex Event Processing* [9](CEP) systems. A CEP system extends the functionality of typical publish subscribe systems while simultaneously maintaining a high through put rate for events. CEP provides operations upon real time event streams such as filtering, correlation, aggregation and event manipulation. In order to supply more expressive subscriptions to that of traditional Publish/Subscribe systems the notion of time and sequencing of events is introduced. As opposed to traditional Publish/Subscribe systems where subscription evaluation is performed only over individual events, CEP systems handle state full subscriptions which can span over multiple events (also referred to in the literature as state full pub/sub). Given the fact that subscriptions maintain state, CEP systems can provide complex operators such as union, aggregation, iteration, negation, etc. The CAYAGA system which is based on finite state automate for evaluating events over a set of complex subscriptions is proposed in the work of [15]. Another example which evaluates complex queries over a stream of data generated from RFIDS is that of SASE [1].

5.1 Content based Publish/Subscribe Systems

5.1.1 Publish/Subscribe systems with an attribute based event model

In the attribute based publish subscribe paradigm subscribers express their interest as boolean predicates over the set of attribute value pairs employed to represent events. Depending on how matching is applied there exist two major categories: *count* based algorithms and *tree* based algorithms

An example of a count based approach for subscriptions adopting conjunctive predicate semantics (featuring equality and inequality predicates), has been presented in [33] and relies on a two phase matching scheme. In the first phase, the set of satisfied predicates is computed via the use of a set of predicate indexes grouping in families predicates with the same attribute and comparison operators. After selecting the set of satisfied predicates by the incoming event, the set of corresponding subscriptions is obtained upon which counting is performed to decide the set of subscriptions that fully match the event. A main drawback of the count based approach is that subscriptions are systematically considered even when some of their predicates are not satisfied.

For this reason [17] proposes a novel indexing scheme that limits the number of subscriptions that need to be evaluated in the two phase matching approach. Instead of counting the returned set of matching subscriptions, a clustering strategy is employed for grouping subscriptions according to their size and common expressions of conjunctive predicates. In addition, the proposed index structure benefits from the cache capabilities of modern processors. Authors devise cost based algorithms for computing the optimal clustering scheme given knowledge about subscriptions and statistics of incoming events.

An example of a Tree based approach for subscriptions adopting conjunctive predicate semantics (featuring equality and inequality predicates), has been presented in [4]. Authors propose a two phase matching scheme and assume a fixed total ordering amongst subscription predicates. In the first phase (pre-processing) the algorithm creates a matching tree built over the subscription predicates based on the considered ordering. Each node is a test of some type and edges are the results to that tests. Each lower level is a refinement of the of the tests performed in higher level. Subscriptions are stored at leafs. When matching (second phase) a top down traversal of the built tree is performed where paths corresponding to successful tests are followed. The set of subscriptions assigned to the leafs the traversal concludes are reported as matched. In the case where subscriptions consist of conjunctions of equality tests of attributes against values, [4] achieves matching time and space complexity that is sub-linear linear correspondingly with respect to the number of subscriptions indexed.

The work of RAPIDMatch [24], which assumes an event model similar to [4], proposes a tree based index that applies a two level partitioning on the set of indexed subscriptions exploiting the fact that in real world applications many events have only a few 'relative' attributes. When evaluating, RAPIDMatch confines it's subscription search space since due to the partitioning applied it can quickly identify a small subset of relevant subscriptions. Finally, the work of [42] presents a multidimensional indexing scheme. The specific work proposes a subscription space dimension transformation that considers events as range queries and subscriptions as points. These multidimensional range queries are evaluated with the use of a UB-Tree index.

A rather different approach to content based filtering that uses Binary Decision Diagrams (BDDs) [10] is presented in the work of [11]. Within that specific approach each elementary subscription predicate is assigned to a boolean variable. Such, subscriptions which combine one or more elementary predicates correspond to boolean functions. Subscriptions are represented

as BDDs and matching is performed via the use of BDD evaluation algorithms. The work of [11] leverages the irrelevance property (although the total number of subscriptions is expected to be large typically only a small fraction of the subscriptions will be interested in a specific attribute value) for efficiently matching events. Additionally representing subscriptions as BDDs exploits subscription predicate commonality between different subscriptions. As opposed to the work of [4] and [17], the use of BDDs allows disjunctive subscription handling rather naturally.

5.1.2 Publish/Subscribe systems with term based event model

All of the work presented thus far supposes a content based subscription schema with a fixed event model (usually small and with small attribute domains) which requires the number of attributes to be defined before hand.

Few works on term based Publish/Subscribe systems have been published in the literature. [37] presents the COBRA Publish/Subscribe system for RSS that crawls, filters, and aggregates vast numbers of RSS feeds, delivering to each user a personalized feed based on their interests expressed as term based subscriptions. The authors consider a distributed environment for matching incoming RSS items against user issued subscriptions. We on the other hand consider a centralized approach and focus on studying how several critical workload parameters influence the bahaviour of both indexing scheme alternatives. The work of [45] conducts a study of several indexing schemes used for the selective dissemination of text documents. The work considers a setting where users submit relatively small user profiles (≈ 5 terms), stored in secodnary memory, which get evaluated against incoming text documents ($\approx 12,000$ terms). More specifically the authors evaluate three different profile indexing schemes: (a) a count based approach as described in section 2.2, (b) a key approach that uses the inverted file for storing a profile in only one of the inverted sets of it's terms (the selected term is called the key and each posting entry contains the profile identifier, the length of the profile and the terms except the key) and (c) a Trie approach as described in section 2.3. In accordance to our work the authors state that Trie requires generally more space than the Count based approach, however, in contradiction to our experimental findings, the authors found that the Trie method requires more time when matching as the increased size in blocks leads to a higher number of I/O's per document. As opposed to their work where they considered intractable studying the analyticaly complexity of the Trie index, we tried to bound the memory requirements of the Trie index based on the statisitcal properties of the vocabulary. Moreover, we observed that the matching time is also influenced by the size of the item. Finally, the authors did not study the impact of different vocabulary distributions on the performance of the indexes. Another example that considers a term based subscription scheme is presented in [22]. The authors consider the counting approach. However, their primary focus in the work is on devising several query processing optimizations for the count index and do not perform a comparative study on other subscription indexing schemes (i.e. tree based). More specifically, [22] exploits term position information and term frequencies for subscriptions and also considers a query clustering technique to further obtain performance benefits.

Finally, in the work of [27], which is the closest to our work, the authors propose a novel indexing scheme in the context of sponsored search. In such a setting advertisers express interest to specific user queries in the form of bids. Whenever a new query is issued, the set of bids that match the query in broad match terms (i.e. every term of the bid is also contained in the query) are returned and displayed. To overcome the limitations of the count index when considering skewed subscription term distributions (i.e. large subscription search space), the authors propose a hash based index built over multi-term combinations of the bids that limits

the search space to only a small fraction of candidate bids. More specifically, the authors use a hash table to index the bids. The value of each entry is a pointer to a so called data node that contains specfic information for the bid (i.e. bid identifier, actual phrase, metadata). In a simple approach they consider indexing the set of terms in each bid (i.e. one data node for every indexed bid). Processing a query requires retrieving entries of data nodes associated to all subsets of the terms in the bid. Apparently, evaluating a query becomes inefficient for large query sizes; the number of probes grows exponentially with query size. To address this specific problem the authors consider a maximum size on the term combinations indexed and propose a mapping scheme that reorganizes bids sharing the same subset of terms to the same data nodes based on a memory access cost model. However, the work addresses the problem of long queries only theoretically (since short queries are the norm) and experimentally evaluates only the simple approach. Apparently, in the setting of web syndication where we expect news items to be much larger than web queries (see section 1.1.3), we cannot apply this simple technique as we would have perform a number of hash table probes equal to the powerset of the item being matched.

5.1.3 Set-valued data indexes

As described in section 2.3, the Trie index requires a total ordering amongst the vocabulary of subscriptions. A rising question is if this serialization for the terms of the vocabulary is actually necessary. A generic indexing scheme for storing set-valued data, based on partial order is POI [41]. In essence, POI is a DAG whose structure reflects the partial ordering of the stored data with respect to the containment (subset/superset) relations that hold. Edges express superset relations; the contents of a particular set stored at a node n is equal to the union of the sets stored in all nodes from n to the top elements of the graph. As opposed to the Trie, POI does not consider serialization and such preserves all of the subset relations that hold. Figure 5.2 illustrates an example built over $\mathcal{S} = \{S_1, S_2, S_3\}$, where $S_1 = \{t_1, t_2, t_3\}$, $S_2 = \{t_1, t_2\}$, and $S_3 = \{t_1, t_3\}$.

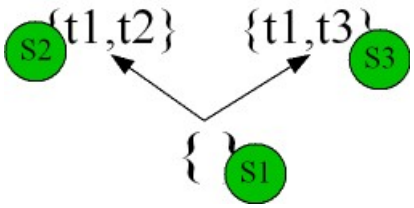


Figure 5.2: POI index overview

Despite the fact that this specific index has been proposed for the compact storage of RDF KB versions (where a RDF KB is a set of triplets, in essence a set of triplet identifiers), the POI index could also be used for our specific problem. Recall that, given a family of sets $\mathcal{S}$ (i.e. subscriptions) and an set I_k (i.e. an item) our goal is to find the sets of $\mathcal{S}$ which are subsets of I_k. More specifically, if we index subscriptions using the POI data structure, then finding the set $\{S_i | S_i \subseteq I_k\}$ could be performed as follows: we can use the proposed insertion algorithm of POI as if we wanted to index I_k; the set of matched subscriptions would then correspond to the parents of I_k. Inserting a subscription S_i requires, (a) finding the direct parents of S_i, (i.e. direct subsets of S_i), (b) finding the direct children of S_i (i.e. the direct supersets of S_i), and (c) linking the new set in the graph. Matching an item I_k on the other requires only step (a) since we are interested in finding the subscriptions that are subsets of I_k and not actually and

not storing it. The authors propose two algorithms for efficiently inserting a set in POI

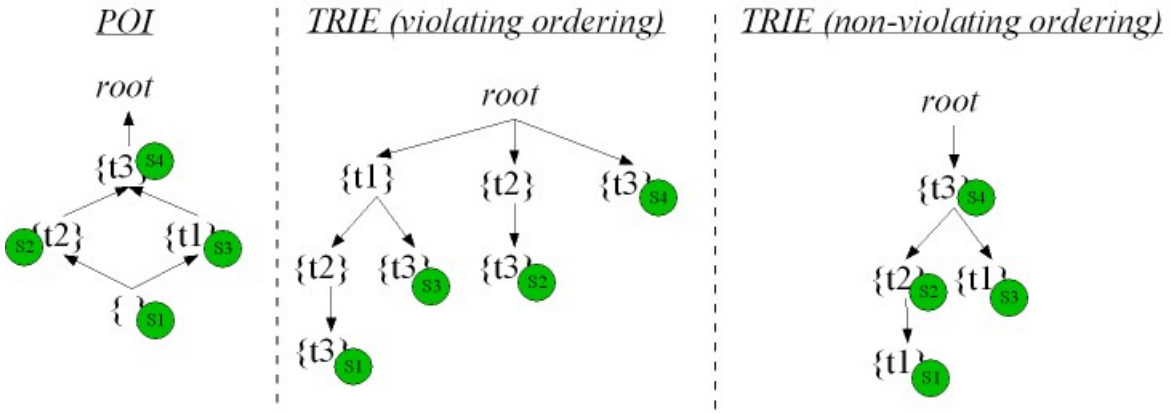


Figure 5.3: POI versus TRIE index

For a comparative discussion of both approaches let us first consider the example indices of Figure 5.3 built over the set $\mathcal{S} = \{S_1, S_2, S_3\}$ of subscriptions where $S_1 = \{t_1, t_2, t_3\}$, $S_2 = \{t_2, t_3\}$, and $S_3 = \{t_3\}$ (subscripts denote ordering). This specific subscription set exhibits two properties: (a) a great deal of subset/superset relations exist, and (b) the ordering considered violates the actual frequency based ordering of the terms (t_3 appears the most times in $\mathcal{S}$ but has the lowest rank). As we can see, this case favours the POI index memory wise. However, if the ordering used for construction did not violate the ordering implied by the actual term frequency distribution, then the two cases would behave similarly (right part of Figure 5.3). Since POI does not require a predefined total ordering amongst the terms of the vocabulary, we can say that it is more robust to vocabulary frequency distribution changes.

POI index performs well when there exists a lot of subset/superset relations amongst the indexed set-valued data and in addition the common subset between them is large. In the context of KB versioning where POI was proposed, this is often the case (versions are large $\approx 10,000$ and new versions are created by adding or deleting a small fraction of elements of a previous version). However if this does not hold (i.e. many partial but not full covering relations exist) then POI would not result to a significant gain. POI exhibits the worst case when no *containment relations* amongst the stored data (i.e. subscriptions) hold. In such a case, POI would become a flat graph and every individual subscription would be stored in a different node. Memory requirements would be equal to the actual size of the subscriptions (no gain), and matching would require examining each individual subscription (in the setting of web syndication we anticipate $\mathcal{S} \approx 10^6$). Apparently, the worst case behaves equally poor to the naive approach (describe in Subsection 2.1) both memory and time wise.

On the other hand, the TRIE index exhibits the worst case when the set of subscriptions do not share any *prefix*. In such a case, as with POI, TRIE memory requirements would be equal to the actual size of the subscriptions (no gain from prefix sharing). With respect to matching, in a worst case TRIE would require to visit a number of nodes equal to all possible subsets up to size $|S_i|$ of the item. In a realistic setting where $|I_k|_{MAX} = 50$ and $|S_i|_{MAX} = 4$ this would be equal to $251,175$, which is much better than the 10^6 set inclusion operations required by POI.

To better demonstrate the contradistinction consider Figure 5.4 which illustrates both structures built over the set of subscriptions of our motivating example of Table 2.1. As we can see, for POI since not many containment relations exist (a) most of the subscriptions are stored individually as is, and (b) in order to determine the set of matched subscriptions it is required to

test almost all of them (naive case). In this context, a more appropriate solution could be to use a semi-lattice representation of the partially ordered covering relations amongst the subscriptions. Since no serialization is applied, this specific approach is independent of the subscriptions' vocabulary frequency distribution and such could result to more compact representations, when compared to TRIE (i.e. when actual ordering violates the ordering used for TRIE construction).

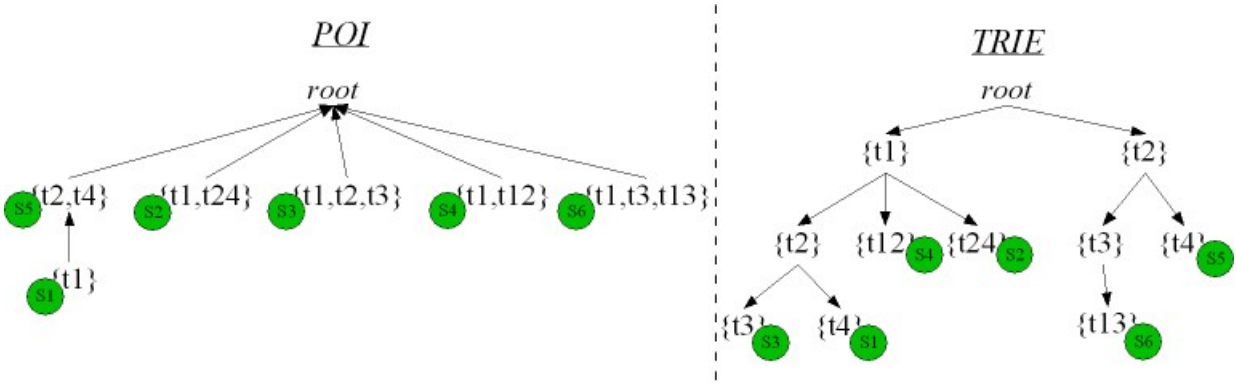


Figure 5.4: POI versus TRIE index

A more extensive comparative and potentially experimental evaluation could be of particular interest for future research. More specifically, it would be interesting to study the performance of POI (memory and time wise) in the specific setting of web syndication, and to determine under what circumstances each approach predominates.

Chapter 6

Conclusions & Future Work

With the continuous growth of online information content syndication has become a popular means for timely information delivery on the Web. However, RSS/Atom technologies as are today, exhibit serious limitations for coping with information overload in the context of Web 2.0 while they imply a tight coupling between feed producers and consumers. In this work we were interested in employing the Publish/Subscribe interaction model for the context of web syndication in order to address both of the limitating factors outlined above. In particular, we considered a Publish/Subscribe framework that enables users to express their interests as keyword based queries which will be matched against incoming news items.

In this work we were interested in studying the behaviour of both count and tree based indices for storing subscrpitions for critical parameters of realistic web syndication workloads. Towards, that end, we presented the well studied Count based index, for key-word based subscriptions, and devised an analytical model that takes into account the distribution of the subscription terms. Based on the model, we showed that under a heavily skewed distribution the Count index's performance heavily degrades and advocated the use of a Trie-based index for storing subscriptions. Additionally, we provided an upper bound for the Trie for both it's memory and matching time requirements. With an extensive set of experimental evaluations, we investigated how several workload parameters impact both indices. We also conducted a thorough investigation on the impact of different subscription term distributions to both indices, a study not yet performed to the best of our knowledge

With respect to the memory required, the Count index outperforms the Trie index in all cases. The memory requirements of the Count index increase linearly when greater subscription and the vocabulary sizes are considered. Interestingly, the impact of the former is of much greater significance than that of the latter. The matching performance for the Count index heavily depends on subscription/news item term distribution. Worst case performance is obtained when both subscriptions and news items follow the same heavily skewed distribution.

The Trie has an excellent matching performance independent to the subscription, vocabulary size and subscription/news item distribution. The matching performance of the Trie outperforms by at least an order of magnitude the Count index in all cases. Despite the fact that it requires more memory than the Count index, by employing path compaction the additional overhead in contradistinction to the performance gain achieved can be considered neglectable. When the actual subscription term distribution follows the distribution based upon which the ordering amongst the terms is applied then the Trie achieves the best performance memory wise.

It is expected for users with same areas of interest to issue similar subscriptions. Given that the Trie compactly stores subscriptions by storing only once shared prefixes, as a future work, we will study how several degrees of similarity amongst the subscriptions impact the performance

of the Trie in comparison to the Count based index. Moreover, we plan to study the problem in a dynamic context where subscription term distributions change in time. Last but not least, we plan to study the more general problem of matching for more complex data models such as the semi-structured one.

Bibliography

[1] Eugene Wu 0002, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *SIGMOD Conference*, pages 407–418. ACM, 2006.

[2] Jian Huang 0002, Ziming Zhuang, Jia Li, and C. Lee Giles. Collaboration over Time: Characterizing and Modeling Network Evolution. In *Proc. Intl. Conf. on Web Search and Web Data Mining (WSDM)*, pages 107–116, 2008.

[3] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[4] Marcos Kawazoe Aguilera, Robert E. Strom, Daniel C. Sturman, Mark Astley, and Tushar Deepak Chandra. Matching events in a content-based subscription system. In *PODC*, pages 53–61, 1999.

[5] Jun-Ichi Aoe, Katsushi Morimoto, and Takashi Sato. An efficient implementation of trie structures. *Softw., Pract. Exper.*, 22(9):695–721, 1992.

[6] Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press / Addison-Wesley, 1999.

[7] Steven M. Beitzel, Eric C. Jensen, Abdur Chowdhury, David A. Grossman, and Ophir Frieder. Hourly Analysis of a Very Large Topically Categorized Web Query Log. In *Proc. ACM Symp. on Information Retrieval (SIGIR)*, pages 321–328, 2004.

[8] Sven Bittner and Annika Hinze. A detailed investigation of memory requirements for publish/subscribe filtering algorithms. In Robert Meersman, Zahir Tari, Mohand-Said Hacid, John Mylopoulos, Barbara Pernici, Özalp Babaoglu, Hans-Arno Jacobsen, Joseph P. Loyall, Michael Kifer, and Stefano Spaccapietra, editors, *OTM Conferences (1)*, volume 3760 of *Lecture Notes in Computer Science*, pages 148–165. Springer, 2005.

[9] Blah. Complex event processing. `http://www.complexevents.com/`, 2008.

[10] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[11] Alexis Campailla, Sagar Chaki, Edmund M. Clarke, Somesh Jha, and Helmut Veith. Efficient filtering in publish-subscribe systems using binary decision. In *ICSE*, pages 443–452. IEEE Computer Society, 2001.

[12] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, 2001.

[13] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel Madden, Vijayshankar Raman, Frederick Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[14] Mark Crovella and Azer Bestavros. Self-similarity in World Wide Web traffic: Evidence and Possible Causes. *IEEE/ACM Trans. Netw.*, 5(6):835–846, 1997.

[15] Alan J. Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker M. White. Towards expressive publish/subscribe systems. In Yannis E. Ioannidis, Marc H. Scholl, Joachim W. Schmidt, Florian Matthes, Michael Hatzopoulos, Klemens Böhm, Alfons Kemper, Torsten Grust, and Christian Böhm, editors, *EDBT*, volume 3896 of *Lecture Notes in Computer Science*, pages 627–644. Springer, 2006.

[16] Yanlei Diao, Peter M. Fischer, Michael J. Franklin, and Raymond To. Yfilter: Efficient and scalable filtering of xml documents. In *ICDE*, pages 341–. IEEE Computer Society, 2002.

[17] Françoise Fabret, Hans-Arno Jacobsen, François Llirbat, João Pereira, Kenneth A. Ross, and Dennis Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In *SIGMOD Conference*, pages 115–126, 2001.

[18] Anindya Ghose and Sha Yang. Analyzing Search Engine Advertising: Firm Behavior and Cross-Selling in Electronic Markets. In *Proc. Intl. World Wide Web Conference (WWW)*, pages 219–226, 2008.

[19] Zeinab Hmedeh. Indexation pour la recherche par le contenu textuel de flux rss. Technical report, CEDRIC Laboratory, CNAM Paris, France.

[20] Zeinab Hmedeh and Cedric du Mouza. Pascal's triangle for modeling the trie index. Technical report, CEDRIC Laboratory, CNAM Paris, France.

[21] Zeinab Hmedeh, Cedric du Mouza, Nicolas Travers, Michel Scholl, Nelly Vouzoukidou, and Vassilis Christofidis. Characterization of RSS Feeds behavior and content: The Roses Testbed. In *preperation*, 2010.

[22] Utku Irmak, Svilen Mihaylov, Torsten Suel, Samrat Ganguly, and Rauf Izmailov. Efficient query subscription processing for prospective search engines. In *USENIX Annual Technical Conference, General Track*, pages 375–380. USENIX, 2006.

[23] Bernard J. Jansen, Amanda Spink, Judy Bateman, and Tefko Saracevic. Real Life Information Retrieval: A Study of User Queries on the Web. *SIGIR Forum*, 32(1):5–17, 1998.

[24] Satyen Kale, Elad Hazan, Fengyun Cao, and Jaswinder Pal Singh. Analysis and algorithms for content-based event matching. In *ICDCS Workshops*, pages 363–369. IEEE Computer Society, 2005.

[25] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[26] Arnd Christian König, Kenneth Ward Church, and Martin Markov. A Data Structure for Sponsored Search. In *Proc. Intl. Conf. on Data Engineering (ICDE)*, pages 90–101, 2009.

[27] Arnd Christian König, Kenneth Ward Church, and Martin Markov. A data structure for sponsored search. In *ICDE*, pages 90–101. IEEE, 2009.

[28] Ryan Levering and Michal Cutler. The portrait of a common html web page. In *Proc. Intl. ACM Symp. on Document Engineering*, pages 198–204, 2006.

[29] Gang Luo, Chunqiang Tang, and Philip S. Yu. Resource-adaptive real-time new event detection. In Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou, editors, *SIGMOD Conference*, pages 497–508. ACM, 2007.

[30] Gilad Mishne and Maarten de Rijke. A study of blog search. In Mounia Lalmas, Andy MacFarlane, Stefan M. Rüger, Anastasios Tombros, Theodora Tsikrika, and Alexei Yavlinsky, editors, *ECIR*, volume 3936 of *Lecture Notes in Computer Science*, pages 289–301. Springer, 2006.

[31] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Singh Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR*, 2003.

[32] Vouzoukidou Nelly. On the statistical properties of web search queries. Technical report, ISL, ICS-FORTH, Greece.

[33] João Pereira, Françoise Fabret, François Llirbat, Radu Preotiuc-Pietro, Kenneth A. Ross, and Dennis Shasha. Publish/subscribe on the web at extreme speed. In Amr El Abbadi, Michael L. Brodie, Sharma Chakravarthy, Umeshwar Dayal, Nabil Kamel, Gunter Schlageter, and Kyu-Young Whang, editors, *VLDB*, pages 627–630. Morgan Kaufmann, 2000.

[34] Bagwell Philip. Fast and space efficient trie searches. Technical report, EPFL Swtzerland.

[35] Bagwell Philip. Ideal hash trees. Technical report, EPFL Swtzerland.

[36] Venugopalan Ramasubramanian, Ryan Peterson, and Emin Gün Sirer. Corona: A high performance publish-subscribe system for the world wide web. In *NSDI*. USENIX, 2006.

[37] Ian Rose, Rohan Murty, Peter R. Pietzuch, Jonathan Ledlie, Mema Roussopoulos, and Matt Welsh. Cobra: Content-based filtering and aggregation of blogs and rss feeds. In *NSDI*. USENIX, 2007.

[38] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. Scribe: The design of a large-scale event notification infrastructure. In Jon Crowcroft and Markus Hofmann, editors, *Networked Group Communication*, volume 2233 of *Lecture Notes in Computer Science*, pages 30–43. Springer, 2001.

[39] Amanda Spink, Dietmar Wolfram, Bernard J. Jansen, and Tefko Saracevic. Searching the Web: The Public and Their Queries. *Jour. of the American Society for Information Science and Technology (JASIST)*, 52(3):226–234, 2001.

[40] Aixin Sun, Meishan Hu, and Ee-Peng Lim. Searching blogs and news: a study on popular queries. In Sung-Hyon Myaeng, Douglas W. Oard, Fabrizio Sebastiani, Tat-Seng Chua, and Mun-Kew Leong, editors, *SIGIR*, pages 729–730. ACM, 2008.

[41] Yannis Tzitzikas, Yannis Theoharis, and Dimitris Andreou. On storage policies for semantic web repositories that support versioning. In Sean Bechhofer, Manfred Hauswirth, Jörg Hoffmann, and Manolis Koubarakis, editors, *ESWC*, volume 5021 of *Lecture Notes in Computer Science*, pages 705–719. Springer, 2008.

[42] Botao Wang, Wang Zhang, and Masaru Kitsuregawa. Ub-tree based efficient predicate index with dimension transform for pub/sub system. In Yoon-Joon Lee, Jianzhong Li, Kyu-Young Whang, and Doheon Lee, editors, *DASFAA*, volume 2973 of *Lecture Notes in Computer Science*, pages 63–74. Springer, 2004.

[43] Hugh E. Williams and Justin Zobel. Searchable words on the Web. *Int. J. on Digital Libraries*, 5(2):99–105, 2005.

[44] Wai Yee Peter Wong and Dik Lun Lee. Implementations of partial document ranking using inverted files. *Inf. Process. Manage.*, 29(5):647–669, 1993.

[45] Tak W. Yan and Hector Garcia-Molina. Index structures for selective dissemination of information under the boolean model. *ACM Trans. Database Syst.*, 19(2):332–364, 1994.

[46] Tak W. Yan and Hector Garcia-Molina. The sift information dissemination system. *ACM Trans. Database Syst.*, 24(4):529–565, 1999.

[47] Jason Y. Zien, Jörg Meyer, John A. Tomlin, and Joy Liu. Web Query Characteristics and their Implications on Search Engines. In *Proc. Intl. World Wide Web Conference (WWW)*, 2001.

[48] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), 2006.